


Università degli Studi di Napoli Federico II

Ph.D. Program in

Information Technology and Electrical Engineering

XXXVIII Cycle

Thesis for the Degree of Doctor of Philosophy

Software Attack Surfaces Reduction via Security Hardening, Fuzzing, and Runtime Enforcement

by

Carmine Cesarano

Advisor: Prof. Roberto Natella

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

To myself, for never giving up, even when the path was unclear and the code refused to compile.

Software Attack Surfaces Reduction via Security Hardening, Fuzzing, and Runtime Enforcement

Ph.D. Thesis presented
for the fulfillment of the Degree of Doctor of Philosophy
in Information Technology and Electrical Engineering
by

Carmine Cesarano

December 2025

Approved as to style and content by

_____

Prof. Roberto Natella, Advisor

Università degli Studi di Napoli Federico II
Ph.D. Program in Information Technology and Electrical Engineering
XXXVIII cycle - Chairman: Prof. Stefano Russo

http://itee.dieti.unina.it

Candidate's declaration

I hereby declare that this thesis submitted to obtain the academic degree of Philosophiæ Doctor (Ph.D.) in Information Technology and Electrical Engineering is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.
Parts of this dissertation have been published in international journals and/or conference articles (see list of the author's publications at the end of the thesis).

Napoli, 3rd December 2025

_____

Carmine Cesarano

Abstract

Modern software systems integrate open-source and off-the-shelf components across multiple abstraction layers. These layers manage complexity through modular stacks of hardware, hypervisors, kernels, container runtimes, and orchestration frameworks. Each component exposes interfaces through which control or data cross trust boundaries. From Kubernetes APIs and inter-process channels to system calls and hypercalls, these boundaries collectively define the attack surface of modern computing systems.

This dissertation addresses how to automatically and precisely reduce attack surfaces across system layers through combined static and dynamic analysis. It introduces novel techniques that infer, enforce, or evaluate least-privilege and minimization principles, either constraining privileges or exposing unsafe behaviors that enlarge the attack surface. Five complementary techniques embody this approach, each targeting a distinct surface.

KubeFence enforces runtime Kubernetes API policies at the fine-grained level of resource specification fields, automatically learning allowed orchestration privileges. *FuzzBox* enables coverage-guided fuzzing of internal communication in closed-source binaries without compiler-time instrumentation, binary rewriting, or hardware tracing. *IRIS* introduces a record-and-replay method to efficiently explore deep hypervisor control flows, enabling targeted mutations at specific internal states to expose complex logic vulnerabilities. *GoSurf* presents a Go-specific taxonomy and static analyzer for supply chain attack vectors, guiding code reviewers toward high-risk components that can be exploited for arbitrary code execution. *GoLeash* prevents import-level supply chain attacks by enforcing allowed privileged capabilities per package rather than application-wide policies.

These techniques demonstrate that automated attack surface reduction is effective across abstraction layers, reducing exploitable interfaces without sacrificing functionality in real-world systems. This dissertation advances the state of the art through a cross-layer methodology that automates assessment and enforcement of exposures, reduces manual effort, and provides a process adaptable to diverse software.

Keywords: Attack Surface Reduction, Security Hardening, Fuzzing, Runtime Enforcement, Software Supply Chain Security.

Sintesi in lingua italiana

I sistemi software moderni integrano componenti open-source e off-the-shelf distribuiti su più livelli di astrazione. Tali livelli gestiscono la complessità attraverso stack modulari di hardware, hypervisor, kernel, container runtime e framework di orchestrazione. Ogni componente espone interfacce attraverso le quali controllo e dati attraversano i trust boundaries. Dalle API di Kubernetes ai canali inter-processo, fino alle system call e hypercall, questi confini definiscono collettivamente l'attack surface dei sistemi di calcolo moderni.

Questa tesi affronta la riduzione automatica e precisa delle attack surfaces nei diversi layers del sistema, combinando analisi statica e dinamica. Introduce tecniche che inferiscono, applicano o valutano i principi di least privilege e minimizzazione, limitando i privilegi oppure individuando comportamenti insicuri che ampliano la superficie d'attacco. Cinque tecniche complementari incarnano questo approccio, ciascuna focalizzata su una specifica superficie.

KubeFence applica policy runtime sulle API di Kubernetes a livello fine-grained dei campi di specifica delle risorse, apprendendo automaticamente i privilegi di orchestrazione consentiti. *FuzzBox* abilita il fuzzing coverage-guided ai canali di comunicazione interna in binari closed-source senza richiedere strumentazione a tempo di compilazione, binary rewriting o hardware tracing. *IRIS* introduce un metodo di record-and-replay per esplorare in modo efficiente i control flow dell'hypervisor, consentendo mutazioni mirate in stati complessi. *GoSurf* presenta una tassonomia e un analizzatore statico specifici per Go per identificare vettori di attacco alla supply chain, guidando le code review verso componenti esposti ad arbitrary code execution. *GoLeash* previene attacchi alla supply chain a livello di import, limitando le capability per ogni package invece di applicare policies globali sull'intera applicazione.

Queste tecniche dimostrano che la riduzione automatica dell'attack surface su più livelli di astrazione è efficace, riducendo le interfacce sfruttabili senza compromettere le funzionalità. La tesi avanza lo stato dell'arte introducendo una metodologia cross-layer che automatizza assessment ed enforcement delle esposizioni superflue, riduce l'effort manuale ed è adattabile a diversi software.

Parole chiave: Attack Surface Reduction, Security Hardening, Fuzzing, Runtime Enforcement, Software Supply Chain Security.

Acknowledgements

The author's work has been supported by a PhD scholarship funded by University of Naples Federico II (UNINA)

In addition, the research presented in this dissertation has been partially supported by:

- Project "GENIO" (CUP B69J23005770005) funded by MIMIT, "Accordi per l'Innovazione" program.
- Project "FLEGREA" (CUP E53D23007950001) funded by MUR PRIN 2022
- Project "IDA—Information Disorder Awareness" funded by the European Union-Next Generation EU within the SERICS Program through the MUR National Recovery and Resilience Plan under Grant PE00000014.
- Project "REINForce: REsearch to INspire the Future" (CDS000609) funded by MISE
- Project "MOST – Sustainable Mobility National Research Center" and received funding from the European Union Next-GenerationEU (PNRR – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.4 – D.D. 1033 17/06/2022, CN00000023).
- Project "COSMIC" funded by DIETI department from UNINA
- Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, by the Swedish Foundation for Strategic Research (SSF)

Contents

List of Acronyms

The following acronyms are used throughout the thesis.

OSS	open-source software
VM	Virtual Machine
IPC	Inter-Process Communication
API	Application Programming Interface
PON	Passive Optical Network
FTTH	Fiber-To-The-Home
OLT	Optical Line Terminal
ONU	Optical Network Unit
DBI	Dynamic Binary Instrumentation
ISA	Instruction Set Architecture
RBAC	Role-Based Access Control
MILS	Multiple Independent Levels of Security
eBPF	Extended Berkeley Packet Filter

List of Figures

List of Tables

Chapter 1

Introduction

The greatest enemy of security is complexity.

Bruce Schneier

Modern software systems are inherently *layered*: hardware, operating systems, middleware, and applications stack together, with virtualization and containerization adding abstraction. Hypervisors isolate Virtual Machines (VMs); container runtimes and kernel primitives enforce lightweight separation; orchestration platforms coordinate distributed workloads. Contemporary stacks also split *control planes* from *data planes*: the former expose declarative Application Programming Interfaces (APIs) and automation hooks to configure, scale, and mutate workloads, while the latter execute computations. This design improves scalability and agility but multiplies opportunities for untrusted input, misconfiguration, or control flow to cross trust boundaries. At these boundaries lie *interfaces*, the connective fabric of layered systems. *External interfaces* (e.g., REST, gRPC, GraphQL) connect clients, operators, and third-party services. *Internal interfaces* (e.g., Inter-Process Communication (IPC) channels, orchestration protocols, message buses) interconnect software components. Concrete boundaries, such as system calls between processes and the kernel, VM exits and hypercalls between a VM and the hypervisor, API requests between services and orchestrators, illustrate exposure points that expand the attack surface. These channels are feature-rich to support flexibility and modularity, but their breadth and configurability create recurring

opportunities for misuse, misconfiguration, and blind spots for security analysis.

A second, equally pervasive trend is extensive reliance on open-source software (OSS) across the stack, from operating systems and container runtimes to orchestration frameworks and application libraries [267]. This model accelerates development and reduces costs but also externalizes control: systems are assembled from community-maintained components whose transitive dependencies and configuration defaults are inherited from upstream. The resulting software supply chains are long and heterogeneous, spanning package registries, build systems, CI/CD pipelines, and container registries, each with distinct trust assumptions. Vulnerabilities and insecure defaults can propagate transitively, and upstream events (dependency takeovers, compromised maintainers, poisoned updates) can ripple through the stack without direct action from downstream developers. In effect, reliance on OSS turns supply chains into another architectural layer with interfaces, dependencies, and trust boundaries that must be secured; otherwise they become an additional attack surface.

Representative examples that embody both multi-layered complexity and heavy reliance on open-source software are contemporary infrastructures implementing the telco-edge paradigm. Traditionally, edge platforms and broadband access networks evolved separately: the former relied on fog servers or resource-limited end devices to run applications, while the latter focused on high-throughput connectivity. Passive Optical Networks (PONs), now the dominant Fiber-To-The-Home (FTTH) technology [97], provide cost-effective, energy-efficient access via a tree topology of Optical Network Units (ONUs) at user premises connected to Optical Line Terminals (OLTs) in central offices. Co-locating computation and storage at OLTs *repurposes* PONs for edge computing, reducing latency and avoiding dedicated fog nodes. The *GENIO* project [60] exemplifies this paradigm: it assembles commodity, OSS-based components into a multilayered edge platform (Figure 1.1), integrating hardware, virtualization technologies, orchestration frameworks, and application workloads that interoperate through external management APIs and internal orchestration protocols. Built on Linux-based systems with Kubernetes as a core backbone, GENIO shows how openness and modularity enable flexibility while expanding the attack surface.

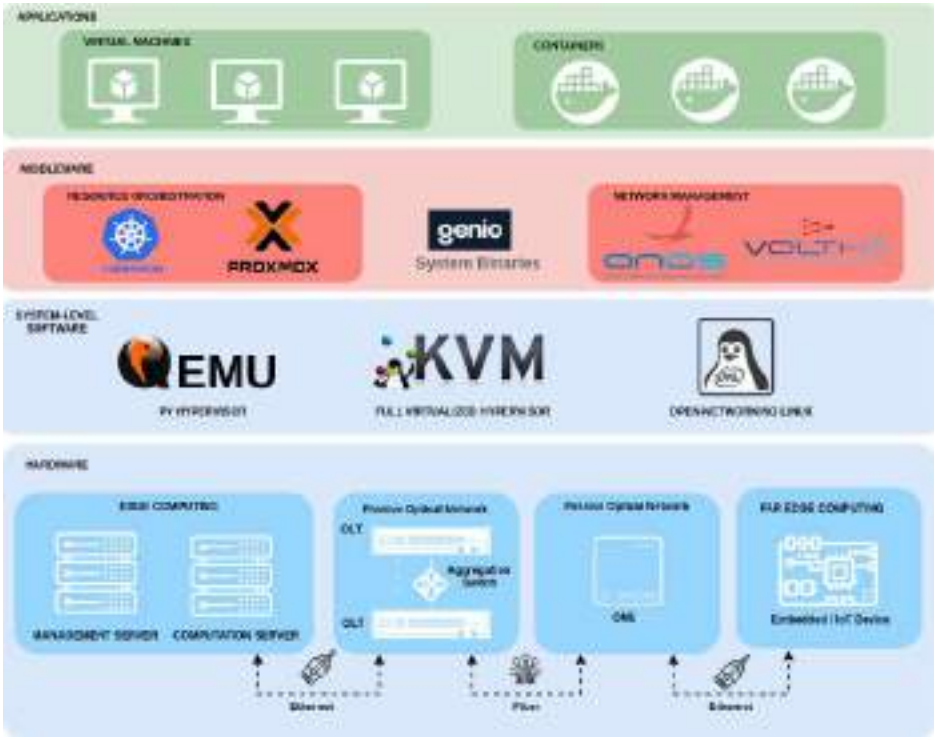


Figure 1.1. Software architecture of the GENIO platform [60].

In summary, the defining properties of modern systems, including layered architectures, rich internal/external interfaces, and open-source supply chains, directly shape today's attack surfaces. Although these attacks surfaces manifest in seemingly different domains (cloud orchestration, embedded IPC, virtualization, and language ecosystems), they are unified by a common abstraction: security-critical interfaces that mediate trust across layers of the same software stack.

In security literature, the *attack surface* is broadly defined as the set of resources, interfaces, and operations exposed to potential adversaries, i.e., the points where an attacker may attempt to enter, influence, or extract data. Recent work [199] emphasizes that attack surfaces are not monolithic but consist of diverse *entry points*, *targets*, and *mechanisms*. For example, OSS-based development accelerates deployment but introduces risks from

unvetted dependencies, while the mixture of external and internal interfaces multiplies entry points that can be misconfigured, chained, or abused. In short, the same features that make infrastructures scalable and flexible also enlarge their attack surfaces.

Modern infrastructures therefore present a broader, dynamic environment to defend, with exposures spanning endpoints, third-party services, and external dependencies [194]. Attacks are often multi-faceted, chaining weaknesses across layers into cross-surface exploit paths. Traditional defenses assume surfaces are isolated, for instance, that securing API endpoints or sandboxing a process suffices. In practice, adversaries increasingly exploit *compositional effects*, combining orchestration APIs, runtime configurations, and supply-chain components. To reason about this complexity, we analyze attack surfaces along four *orthogonal axes* that recur throughout the thesis:

- **Orchestration surface**: Resource orchestrators such as Kubernetes expose thousands of API verbs and fields. Their breadth enables privilege escalation and lateral movement, as shown by attacks exploiting malicious API requests or misconfigured role-based access control [63].
- **IPC surface**: Inter-process communication mechanisms, from sockets and system calls to message buses, are opaque, stateful, and difficult to test. Unsafe serialization, handler misuse, and weak access controls frequently reappear as entry points [199].
- **Hypervisor surface**: Hardware-assisted virtualization introduces boundaries such as hypercalls and VM exits, which underpin multi-tenancy isolation but are repeatedly targeted in privilege-escalation CVEs [59].
- **Software supply chain surface**: Open-source ecosystems introduce risks via unvetted dependencies, malicious packages, and opaque third-party code. Studies [194] note that organizations often underestimate external attack surfaces, including suppliers and partners.

These axes are *orthogonal* in that they reflect distinct classes of interfaces, attacker entry points, and defense strategies. Yet they are not

independent in practice: for instance, an orchestration API call may internally rely on IPC mechanisms or trigger a hypercall under the hood. We therefore treat them as *analytically orthogonal* but *operationally composable*: securing one surface does not eliminate exposure if others remain open.

Many defenses exist, including hardening, testing, and runtime enforcement, across multiple layers. In orchestration, pre-deployment static analysis [303, 297, 257, 208, 132, 51, 84, 190, 196, 178, 128], container-centric runtime controls [158, 153, 307, 71, 104], and REST-API hardening [187, 148, 40, 154] raise the baseline. Yet they lack *workload-specific, fine-grained* runtime policy enforcement for deeply nested configurations. For industrial IPC surfaces, static binary rewriting [9, 87, 92] and Dynamic Binary Instrumentation (DBI) [8, 11] provide coverage guidance primarily on mainstream Instruction Set Architectures (ISAs) and on user space targets. Hardware-assisted tracing [66, 6, 253, 182, 98] offers precision but ties solutions to specific architectures. User-mode emulation [7, 109, 188] scales, but cannot exercise full-system images, leaving opaque, multi-partition deployments under-tested. At the hypervisor layer, formal methods [161, 112, 181, 56, 47, 138] provide assurance but struggle to scale to realistic hardware models. Prior CPU/device fuzzers [32, 299, 111, 115, 251, 252, 139, 230, 55, 200] often depend on manual seeds, grammars, or heavy instrumentation, limiting exploration of *stateful, guest-driven* paths. In the software supply chain, ecosystem-agnostic taxonomies [218, 176, 177], vulnerability scanners [226, 264], and rule-based linters [224, 225, 227, 124] aid pre-deployment vetting. However, they miss *language- and lifecycle-specific* execution vectors. Dynamic sandboxes and package-level systems improve containment, but frequently rely on manual policies and lack *per-package, syscall-level* enforcement resilient to obfuscation.

Collectively, these gaps materializes into five recurring challenges: (1) *Granularity*: Mechanisms such as Kubernetes Role-Based Access Control (RBAC) and container seccomp operate at coarse levels, lacking the fine-grained control needed to constrain specific API fields or package-level behaviors. (2) *Opacity of closed systems*: Industrial and embedded platforms often ship as opaque binaries with proprietary toolchains, preventing instrumentation and limiting visibility for testing and enforce-

ment. (3) *Encrypted and obfuscated execution*: Malicious behaviors hide within encrypted channels, runtime reflection, or delayed execution, evading static analysis and shallow monitoring. (4) *Dynamic and stateful complexity*: Hypervisor interfaces, IPC protocols, and orchestration APIs exhibit highly stateful and nested behaviors that are difficult to explore or constrain with static rules or shallow testing. (5) *Supply chain specificity*: Open-source ecosystems, such as Go, introduce language- and lifecycle-specific attack vectors that generic taxonomies and tools fail to capture, leaving ecosystem-tailored risks unaddressed.

These difficulties motivate the need for new strategies. The central research problem of this dissertation is therefore: *How can the attack surface of complex, layered systems be reduced in a way that is automated, fine-grained, and resilient against evasion across multiple layers of abstraction.* Addressing this problem requires approaches that combine static and dynamic methods, are tailored to the unique properties of each layer, and share common goals: automation, fine granularity, and resilience against obfuscation.

This dissertation advances the state of the art in attack surface reduction by introducing five complementary contributions, each targeting a distinct *interface layer* of modern software systems, from cloud orchestration APIs to low-level IPC, hypervisor boundaries, and language-level supply chains:

(i) **KubeFence** [63]: a runtime enforcement system for Kubernetes APIs. Unlike RBAC or container-centric defenses, KubeFence derives workload-specific policies dynamically and enforces them at runtime. Its fine-grained analysis of nested API request structures makes it the first system to provide automated debloating of orchestration layer operations. Evaluation shows that KubeFence reduces exploitable Kubernetes API surfaces while preserving legitimate workload functionality.

(ii) **FuzzBox** [62]: the first full-system emulation-based fuzzer for IPC mechanisms in binary-only, industrial systems. Unlike instrumentation-based fuzzers, FuzzBox requires no source code, toolchain, or hardware dependencies. By integrating coverage-guided fuzzing into QEMU execution, it enables input mutation, fault detection, and state ex-

ploration in closed environments. Experiments on a Multiple Independent Levels of Security (MILS) hypervisor and IoT firmware show portability and effectiveness for testing previously unreachable components.

(iii) **IRIS** [59]: a hypervisor fuzzing framework that combines record-and-replay with mutation-based testing. Prior methods struggle to reproduce realistic VM states or require handcrafted seeds. IRIS captures guest-driven execution traces, replays them deterministically to restore complex states, and applies mutations to explore under-tested CPU virtualization paths. A prototype shows accurate state reproduction and efficient fuzzing of hypercalls, revealing otherwise unreachable vulnerabilities.

(iv) **GoSurf** [58]: a static analysis tool implementing the first Go-specific taxonomy of supply chain attack vectors. It maps 12 execution vectors across pre-build, initialization, and runtime phases of Go packages. Unlike generic scanners or linters, GoSurf systematically detects language- and ecosystem-specific risks in dependency graphs and call structures. An evaluation on 500 packages demonstrates its ability to surface previously overlooked supply chain risks.

(v) **GoLeash** [61]: the first runtime enforcement system for Go that applies least privilege at the package level. Unlike coarse process- or container-level sandboxes, GoLeash automatically generates syscall policies per package and enforces them dynamically. It constrains obfuscated or delayed malicious behaviors introduced through dependencies. Experiments show that GoLeash blocks supply chain attacks with acceptable runtime overhead, effectively addressing evasive threats in Go binaries.

The remainder of this thesis is organized as follows:

Chapter 2 surveys background and related work on attack surfaces and reduction strategies, positioning the thesis contributions.

Chapter 3 analyzes the Kubernetes orchestration surface and presents KubeFence for runtime, workload-aware API debloating.

Chapter 4 examines opaque IPC surfaces in closed systems and introduces FuzzBox for emulation-based, grey-box fuzzing.

Chapter 5 explores hypervisor attack surfaces and develops IRIS for record-and-replay guided fuzzing.

Chapter 6 investigates Go supply chain static risks and introduces a taxonomy of attack vectors and GoSurf for taxonomy-driven analysis.

Chapter 7 addresses Go supply chain runtime risks and presents GoLeash for fine-grained policy enforcement.

Chapter 8 discusses broader implications, open challenges, and future research directions in cross-layer attack surface reduction.

Chapter 2

Attack Surfaces in Modern Software Systems

In computer security, a *vulnerability* is a flaw or weakness in a system that can be exploited to violate fundamental security properties such as confidentiality, integrity, or availability. An *attack* is the concrete realization of such an exploitation through one or more adversarial actions, while *security mechanisms* aim to prevent, detect, or mitigate these attacks.

Building on these notions, this chapter provides the foundation for understanding the diverse attack surfaces in modern software systems, common attack patterns, and existing defenses to highlight key limitations of the state of the art and to position our contribution in each domain.

2.1 Attack Surfaces

The notion of the *attack surface* has become a central abstraction for reasoning about system exposure and adversarial opportunities. Rather than treating it as a single, static boundary, we emphasizes that modern infrastructures expose multiple, evolving surfaces shaped by architectural layering, pervasive automation, and reliance on open-source ecosystems. As a result, defending today's systems requires understanding not only which resources are reachable but also how diverse interfaces interact and compose into exploitable chains. In this section, we provide a structured overview of the principal axes of attack surfaces that recur across con-

temporary software. Each axis highlights a distinct set of interfaces and risks. By analyzing each axis in turn, we aim to position our contributions against the broader literature on attack surface reduction.

2.1.1 Orchestration Surface

Modern cloud-native and edge computing deployments rely on orchestration systems (e.g., Kubernetes [168], OpenShift [240], Nomad [136]) to coordinate containerized workloads. These systems expose a privileged management API that enables workload configuration, scheduling, scaling, service discovery, and failure recovery.

Kubernetes. Kubernetes (K8s) [168] has become the mainstream platform for orchestrating containerized applications, enabling scalable deployment and management across distributed environments. Its adoption spans various critical domains, including finance, healthcare, and government services [72, 172], where security is crucial. Kubernetes abstracts underlying physical resources into logical entities called *Kubernetes resources*, which include Pods (for workloads), Services (networking), Volumes (storage), ConfigMaps (configuration), and Secrets (security). These resources are managed within a cluster, consisting of worker nodes orchestrated by a control plane. The K8s API Server is the central management component, responsible for handling operations withing the cluster. It exposes a RESTful API that allows users to interact with the cluster by sending HTTP requests to create, modify, and delete resources. The API Server supports HTTP verbs like *get*, *post*, *put*, and *delete* to manipulate Kubernetes resources. Each resource is represented as a *Kubernetes Object*, typically defined using a declarative manifest file in YAML or JSON format. The manifest specifies the desired state of a resource, which the K8s control plane works to reconcile with the current state. Kubernetes Objects generally contain two primary nested properties, which are `spec` and `status`, to describe the desired and current state, respectively. This structure is typical for objects like Pods, Deployments, and Volumes, as shown in Figure 2.1.

An example of a desired state is to bring up a given number of container replicas. Objects like Secrets and ConfigMaps omit these fields, focusing instead on storing sensitive data or configuration in the `data` field. The

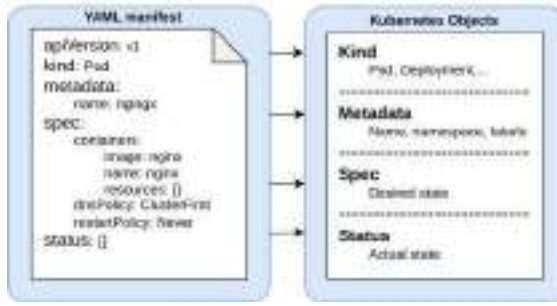


Figure 2.1. Example of YAML Manifest to configure a K8s Object [63].

K8s API exposes a set of endpoints for each resource. When a user defines a resource in a manifest, specifies all configurable fields, and applies this configuration to the cluster, an HTTP request is triggered to the relevant API endpoint, including all the resource configurations in the payload. This workflow allows users to configure resources declaratively by specifying their intent, while the API server translates these configurations into actual cluster state changes. However, these exposed API endpoints and configurable fields contribute to a significant attack surface.

Kubernetes RBAC. To mitigate security risks, K8s provides an RBAC access control mechanism [167]. RBAC defines which users, groups, and service accounts can perform specific actions on resources, such as viewing, creating, modifying, and deleting them. RBAC policies are defined using YAML manifests through four kinds of Kubernetes Objects: *Role*, *ClusterRole*, *RoleBinding*, and *ClusterRoleBinding*. A Role or ClusterRole object contains rules that specify a set of permissions, while RoleBinding and ClusterRoleBinding objects grant the permissions defined in a role to a user or set of users. This is particularly relevant in multi-user K8s clusters, where developers should be restricted to working with designated objects, preventing them from accessing others.

While RBAC provides a structured approach to access control, it has limitations in terms of granularity. RBAC policies do not inspect the contents of K8s resource specifications in input to the API. This means that even if a user is restricted to only access specific K8s resources, they can still abuse or exploit all of the features available for that resource.

Therefore, a more granular enforcement mechanism capable of inspecting and filtering API requests at a deeper level is needed to block potential misconfigurations and exploits. This limitation also explains why misconfigurations and API-level vulnerabilities often bypass high-level access controls.

K8s Operators and Helm templates Kubernetes relies on *controllers* to continuously monitor and reconcile the desired and current states of cluster resources. Built-in controllers are preconfigured to handle standard Kubernetes resources and provide basic features, such as autoscaling and self-healing. However, for more complex operations that extend beyond the capabilities of built-in controllers, users must adopt custom controllers. These ad-hoc controllers, known as *Kubernetes Operators [166]*, are K8s API clients that automate advanced lifecycle management tasks for stateful and specialized applications. Operators continuously monitor and adjust the application state in a control loop. For instance, if the user specifies that an application should maintain three replicas, the Operator constantly checks the cluster state. If it detects that one replica has failed, it automatically triggers a new deployment to restore the desired count. This capability allows operators to handle both Day-1 (installation, configuration) and Day-2 (updates, scaling, monitoring) operations, reducing manual intervention.

Operators can be implemented in various ways, using the Go language, Ansible, and Helm [131]. Among these, Helm-based operators are by far the most common. The widespread adoption of Helm is evident in catalogs such as Artifact Hub [73] and OperatorHub [239], which lists hundreds of Helm-based Operators already distributed for production use, spanning applications such as databases, monitoring tools, and CI/CD pipelines.

Helm is the de facto package management solution for K8s [303], simplifying the process of handling complex K8s resources to package, configure, and deploy applications. Helm packages, known as *charts*, provide `templates` files created by chart developers. These templates define K8s resources needed to run an application. They consist of fixed parts and placeholders for configurable values, as shown in Figure 2.2. Defaults for the configurable values are typically included in a separate `values` file, in order to provide an initial configuration for the K8s resources. Users of

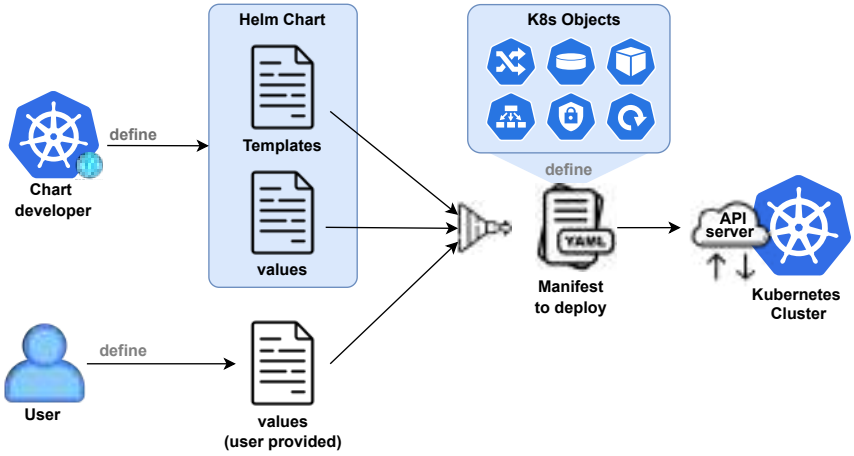


Figure 2.2. Helm Template Processing [63].

the K8s operator can customize and override to meet specific workload requirements. This flexibility allows users to deploy the same application across different environments with minimal changes [140].

When deploying an application, Helm processes the template by combining them with values to generate complete K8s manifests. Beyond simple value assignment, Helm templates support advanced conditional logic through directives such as `if-else` or `range`. These directives enhance template flexibility, enabling developers to include or exclude fields, iterate over collections, or conditionally populate fields based on the provided values (e.g., enabling optional configurations, as shown in Figure 2.3). These manifests are then submitted to the K8s API Server to create or update resources. In practice, Helm templates constrain the inputs that are sent to the K8s API Server, since the user does not change the fixed parts of the templates.

From a security standpoint, this duality is important: while Helm templates reduce free-form user input (potentially mitigating malformed requests), they also expand the attack surface by introducing large volumes of third-party templates into clusters. Each template or Operator becomes trusted code capable of generating privileged API requests, thereby multiplying the pathways an attacker may attempt to exploit. The flexibility

```
apiVersion: v1
kind: Secret
metadata:
  name: {{ template "mlflow.fullname" . }}-env-secret
  labels:
    dict: Dict
    app: {{ template "mlflow.name" . }}
    chart: {{ template "mlflow.chart" . }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
type: Opaque
data:
  dict: Dict
{{- if .Values.backendStore.postgres.Enabled }}
  PGUSER: {{ .Values.backendStore.postgres.user }}
  PGPASSWORD: {{ .Values.backendStore.postgres.password }}
{{- end }}
```

Figure 2.3. Helm *Template* for a *Secret* resource.

and extensibility of K8s, while providing significant advantages for deployment and scaling, also introduce substantial risks. The attack surface exposed by the K8s APIs is particularly concerning, as *malicious API requests* can abuse features and exploit vulnerabilities of K8s. We categorize security risks related to the K8s' attack surface in two broad categories.

Misconfigurations of a K8s cluster K8s is not inherently secure by default. Proper configuration is crucial to maintaining a secure environment, but the complexity of this task often leads administrators to prioritize ease of deployment over rigorous security practices. This can result in security misconfigurations that inadvertently weaken the security of the cluster [208]. For example, running containers with elevated privileges or misapplying resource limits can expose critical resources and escalate privileges. Empirical studies [238] have shown that common configuration errors, such as overly permissive network policies or default access settings, can leave clusters vulnerable to exploitation.

Malicious users can leverage these misconfigurations to abuse the cluster. For example, if the cluster runs containers with high privileges, and the user omits the `runAsNonRoot` specification attribute, the user can escalate privileges. Another example is to leave service accounts enabled (e.g.,

`defaultServiceAccount`), which provides the user with permission to access the K8s API in every namespace. Finally, some functionalities require careful configuration to avoid introducing weaknesses. For example, inadequate TLS/SSL settings can expose communication channels to interception.

These issues often do not stem from flaws in K8s itself, but rather from how the system is configured by administrators. When misconfigured, a K8s cluster can quickly become an attractive target for attackers. It is also important to note that RBAC does not provide control over potential abuses of such features, since it does not inspect parameters within resource specifications.

```
apiVersion: v1
kind: Pod
spec:
  initContainers:
  - name: busybox
    image: "busybox"
    command: ["ln", "-s", "/", "/mnt/data/symlink-door"]
    volumeMounts:
    - name: test-vol
      mountPath: /test
  containers:
  - name: my-container
    image: "ngingx"
    volumeMounts:
      - mountPath: /test
        name: my-volume
        subPath: symlink-door
  volumes:
  - name: my-volume
    emptyDir: {}
```

Figure 2.4. Malicious K8s API request triggering CVE-2017-1002101.

Software Vulnerabilities in the K8s codebase Beyond misconfigurations, K8s itself is susceptible to vulnerabilities in its codebase. Numerous CVEs have been found in the recent past, and more are likely to occur in the future due to the complexity of the K8s project. Some of these CVEs can be directly exploited through the K8s API, by injecting malicious input values in object specifications. These exploits can cause disruption of

cluster operations, privilege escalation, and unauthorized access to sensitive data.

For example, in K8s clusters prior to version 1.9.4, the vulnerable `subPath` feature can be exploited by attackers to access sensitive directories on the host filesystem (CVE-2017-1002101). As illustrated in Figure 2.4, this vulnerability can be exploited by sending a maliciously crafted API request to deploy a K8s Pod. Specifically, an init container creates a symbolic link to the root directory of the host, and the main container then mounts this symlink as a `subPath`, granting access to the host filesystem.

This is only one example, but dozens of CVEs have targeted K8s API components [63]. Collectively, these vulnerabilities demonstrate that restricting user permissions at a high level, as does RBAC, is insufficient. Limiting access to certain resources (e.g., by denying access to other resources except Pods) does not prevent attackers from manipulating specific configuration parameters of unrestricted resources to exploit underlying vulnerabilities. The limitations of RBAC are intrinsically due to its conceptual model, which is coarsely defined around "roles" and "resources," since it is designed for manual definition and review by administrators. Even if a finer-grained RBAC existed, it would introduce excessive complexity for this use case. Thus, an automated and more detailed filtering of parameters within API requests is necessary to reduce the orchestration attack surface.

Orchestration Interfaces as an Attack Surface

Kubernetes exemplifies the orchestration-layer attack surface: a highly privileged API plane where complexity, unpatched vulnerabilities, and misconfigurations collectively expand the adversarial attack vector for control over the orchestrated environment. Unlike isolated software flaws, this surface is systemic, emerging from the breadth of exposed operational primitives and the inherent difficulty of enforcing fine-grained access and authorization controls. These characteristics make orchestration APIs a persistent entrypoint in modern exploit chains, underscoring the need for precise enforcement mechanisms discussed later in this thesis.

2.1.2 Inter-Process Communication Surface

Modern complex systems rely heavily on **inter-process communication** (IPC) to coordinate services, exchange state, and manage control flows between components. IPC mechanisms provide structured communication channels such as shared memory, sockets, and message queues, which act as the binding layer that integrates subsystems. Examples range from sockets used by system daemons to control endpoints embedded in industrial devices and shared-memory regions in IoT platforms. While essential, IPC channels often represent *hidden attack surfaces*. Unlike public-facing APIs, they are typically considered "internal" to the platform and therefore: (1) are poorly documented or entirely undocumented; (2) are inconsistently authenticated, with many assuming local clients are implicitly trusted; (3) receive little scrutiny from traditional security testing. As a result, vulnerabilities in IPC pathways can become critical vectors in exploit chains, enabling attackers to escalate privileges, bypass isolation guarantees, or compromise components assumed to be trustworthy. Concretely, IPC frequently coincides with trust boundaries: container and cloud runtimes expose CRI endpoints and daemon control sockets; OS service buses such as D-Bus mediate requests between unprivileged clients and privileged daemons; distributed brokers (MQTT, NATS, AMQP) bridge microservices and IoT devices; and in high-assurance environments, partitioned systems rely on controlled channels to enforce strict domain separation. In all these cases, IPC carries privileged control or sensitive data across boundaries, so small authentication, policy, or parsing mistakes can yield high-impact compromise. These risks are exacerbated in industrial embedded systems, which often rely on proprietary or outdated toolchains and heterogeneous configurations, limiting both security observability and the applicability of traditional defenses.

MILS-based Systems. An illustrative case is represented by MILS (Multiple Independent Levels of Security) systems [30], such as Wind River VxWorks [204]. MILS is a high-assurance security architecture, based on the concept of separation and controlled information flow. Its core abstraction is the "*separation kernel*", which can be viewed as a specialized hypervisor providing strict isolation between applications of different security levels running on different "*partitions*" and allowing them to coexist

on the same hardware platform. In these systems, IPC is not an auxiliary feature but the only channel across isolated partitions. Cross-partition communication is realized through Secure Inter-Partition Communication (SIPC) channels [301], provided and mediated by the MILS kernel. At the low level, SIPC channels are explicitly configured at build time, with each channel associated to a pair of sender and receiver endpoints bound to specific partitions. Access to a channel is performed through system calls to the MILS kernel, which copy or enqueue messages into a kernel-managed buffer. Messages are queued and delivered according to strict scheduling policies, typically in bounded-size buffers with deterministic behavior to preserve real-time guarantees. Synchronization between partitions is enforced by the kernel, either through blocking send/receive semantics or via notification mechanisms when data becomes available. Thus, SIPC channels are tightly controlled, kernel-mediated constructs, but because they are the sole mechanism for inter-partition communication, they become a uniquely concentrated attack surface. The security implications are particularly significant given that MILS-based separation kernels are adopted in safety- and security-critical domains such as automotive, railways, avionics, and defense [204].

MILS-Based Gateway. A relevant use case for MILS technology is the *MILS-based gateway* [203], which mediate traffic between network domains of differing criticality, such as a "*sensitive*" and an "*unrestricted*" network [28]. An example setup implementing this communication mechanism is shown in Figure 2.5.

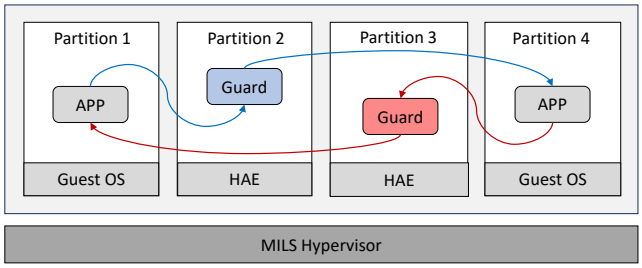


Figure 2.5. Multiple Independent Levels of Security architecture [62].

Four virtual boards (partitions) are deployed: a low-criticality par-

tition connected to the unrestricted domain; a high-criticality partition connected to the sensitive domain; a *guard* partition that inspects and filters traffic flowing from the low domain to the high domain; a second *guard* partition that mediates traffic in the opposite direction, from high to low. In this use case VxWorks Guest OS, a Linux-based kernel is used for low-criticality partitions and the High Assurance Environment (HAE), a minimal run-time library is used for high-criticality partitions. Each guard is connected to its neighboring partitions through SIPC channels managed by the separation kernel. Two SIPC channels handle low → guard → high flows, while another two handle high → guard → low flows. This design enforces the principle that all cross-domain communication must traverse a dedicated guard. However, it also highlights why SIPC channels are a critical *attack surface*: they directly interconnect partitions of different criticality. An attacker compromising the low domain can inject malformed inputs over the SIPC channel to the low→high guard, targeting its parsing and filtering logic. Similarly, flaws in the high→low guard could enable data leakage or unauthorized downgrading of sensitive information. In both directions, the guards themselves become security-critical components whose compromise undermines the entire isolation model.

Design and Configuration Risks in Guards. Even though MILS architectures are engineered for high assurance, their security still depend on the correct design and configuration of guards and SIPC channels. A misconfigured SIPC channel, for example, one that allows overly broad data flows or fails to enforce message size and type constraints, can undermine isolation guarantees and enable unsafe cross-domain communication. Likewise, guard policies must be precise in filtering and validating inputs; any oversight may result in privilege escalation or data leakage. In practice, these risks are compounded by the opacity of the MILS toolchain. Configurations are generated through proprietary environments such as the VxWorks Workbench [20], where system integrators specify processor type, memory layout, device mappings, SIPC channels, and security policies. The resulting system image encapsulates all partitions and guards in a single binary, leaving no hooks for instrumentation, coverage analysis, or runtime tracing. This lack of transparency makes it difficult to verify whether SIPC and guard policies are correctly enforced. Thus,

misconfiguration in MILS systems is not only a matter of human error but also of limited visibility, amplifying the risk that unsafe IPC pathways remain undetected.

Vulnerabilities in Guard Implementations. Beyond configuration, the guard components themselves may contain software vulnerabilities. Guards are typically implemented as user-space applications within high-assurance partitions, but they must parse and inspect arbitrary inputs from less trusted domains. This parsing logic can harbor memory safety bugs, improper validation, or logic flaws. An adversary can compromise a low-assurance partition, such as the Linux guest connected to the unrestricted domain in a MILS-based gateway. From this vantage point, the attacker can send arbitrary messages into SIPC channels bound to guards, either as individually or as sequences, effectively reaching the trusted guard code that enforces isolation, and attempting to trigger potential vulnerabilities. Previous work [44] on protocol parsers and input validation components in other domains shows that such code is highly error-prone. In the MILS context, a single flaw in a guard implementation undermines the separation guarantees of the entire architecture, allowing an attacker to escalate across partitions or exfiltrate sensitive information. Beyond parser bugs, additional classes of flaws are possible: *protocol state-machine vulnerabilities*, where unexpected message sequences drive guards into untested or invalid states; *policy bypasses*, where incomplete enforcement enables data to slip through filtering logic; and *resource exhaustion*, where flooding SIPC channels degrades scheduling and availability guarantees. This diversity highlights the depth of potential attack vectors reachable solely via IPC.

IPC as an Attack Surface

In MILS-based systems, guards are the trusted components responsible for mediating and filtering all cross-domain communication. The robustness of these guards directly determines whether the isolation guarantees of the MILS architecture hold in practice. Although guards runs within internal partitions, they remain reachable through the SIPC channels that connect them with low- and high-criticality domains. This means that adversaries controlling a low-assurance partition can inject crafted or malicious inputs into the guards via SIPC, attempting to exploit flaws in their parsing, state handling, or filtering logic. Consequently, IPC mechanisms in MILS systems represent a concentrated *attack surface*.

2.1.3 Hypervisor Surface

Virtualization has become the cornerstone technology across both cloud computing and safety-critical domains, primarily thanks to its ability to consolidate multiple workloads on the same hardware platform. Its adoption is also driven by the need for strong isolation guarantees, which are essential in industrial and safety-certified systems.

Virtualization and Hypervisors. Virtualization underpins modern computing, spanning from large-scale cloud platforms to safety-critical industrial domains such as avionics, automotive, and defense [70]. This is because of its ability to reduce *SWaP-C* factors (size, weight, power, and cost) by consolidating multiple software stacks on a single system-on-a-chip (SoC) [70, 1]. In addition, its isolation properties [236] make it an enabling technology for *Industry 4.0* [78, 67] and a key requirement in safety standards (e.g., DO178C for avionics [247], ISO 26262 for automotive [143]), which demand evidence of temporal, memory, and fault isolation. Violations of these guarantees can undermine safety and lead to catastrophic consequences [180]. Virtualization allows dividing the hardware resources of a computer into multiple *virtual* computers, called Virtual Machines (VMs) or *guests*. It is implemented by a software layer, the *hypervisor*, which abstracts physical CPU, memory, and I/O, to run different and isolated application environments, including the operating system (OS). In

particular, *hardware-assisted virtualization* takes advantage of CPU virtualization technologies (e.g., Intel VT-x [202], AMD-V [31], ARM VHE [81]) to implement *full virtualization*, i.e., emulating a complete machine to run unmodified guests. These hardware extensions introduce a novel (and higher) level of privilege for the hypervisor. This way, developers can implement *virtual CPU (vCPU)* abstractions that can run a guest OS at a lower level of privilege compared to the hypervisor. If the guest OS needs to execute a *sensitive* instruction (e.g., page table update, interrupt handling, etc.), the execution traps, and the control is passed to the hypervisor.

Guest-Hypervisor Interactions. The hypervisor exposes a set of privileged interfaces that allow guest systems to interact with it:

- **VM exits**: implicit transfers of control from guest to hypervisor when a sensitive instruction is executed or a hardware event occurs.
- **Hypercalls**: explicit API calls that allow guest operating systems to request hypervisor services, such as memory management or device operations.
- **I/O traps and scheduling events**: redirections of guest access to devices or resources, enabling the hypervisor to mediate I/O and CPU scheduling.

All these interfaces are available to guest workloads, including potentially malicious ones, and each trigger switches to the hypervisor's most privileged context. Their robustness therefore directly determines whether the hypervisor can enforce isolation.

A representative example of these interfaces in action is Intel VT-x (as depicted in Figure 2.6). When virtualization is enabled via the `VMXON` instruction, processors introduce two execution modes: root mode for the hypervisor and non-root mode for guest VMs. These modes are orthogonal to traditional CPU privilege rings and operating modes (real, protected, long mode). Then, the hypervisor configures each virtual CPU through a Virtual Machine Control Structure (VMCS). The VMCS contains the guest and host register states, control fields defining which operations should trigger exits, and an exit information area recording metadata about the most recent transition. Access to the VMCS requires privileged VMX instructions (`VMREAD`, `VMWRITE`); bypassing them risks undefined hardware

behavior [141]. Guest execution begins when the hypervisor loads the VMCS and issues `VMLAUNCH`. From this point forward, the guest runs in non-root mode almost as if on bare metal, except that specific privileged instructions or events transparently trigger a transfer of control back to the hypervisor. Such a transfer is called a *VM exit*. Each VM exit involves a hardware context switch: guest state is saved to the VMCS, host state is restored, and execution resumes in a hypervisor-defined exit handler. Once the handler completes its work, control is returned to the guest via `VMRESUME`. Intel's architecture specifies 69 distinct exit reasons (Appendix C, Table 1-c [141]), covering privileged instructions (`RDMSR`, `WRMSR`), control register access, interrupts, and I/O operations. Hypervisors may also configure additional exits, for example to implement memory deduplication [189] or VM introspection [137].

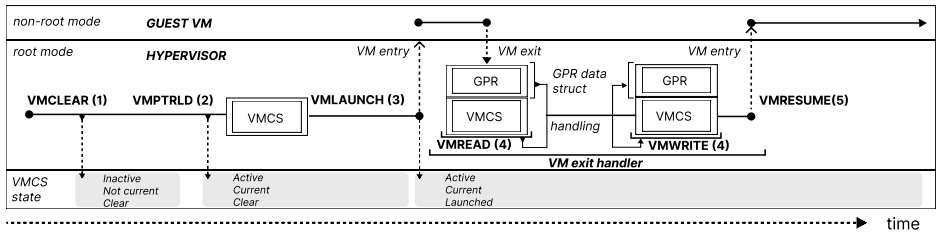


Figure 2.6. Workflow of a virtual machine in Intel VT-x [59].

Vulnerabilities in VM Exit Handlers. The number and diversity of VM exits make exit handlers both performance-critical and security-sensitive. Every VM exit represents an untrusted-to-trusted control transfer: guest-controlled inputs (registers, VMCS fields) directly influence hypervisor logic executed in its most privileged context.

To illustrate this complexity, consider the Xen hypervisor [45] handling a guest switch into protected mode. This transition requires carefully ordered steps: clearing interrupts, preparing the Global Descriptor Table (GDT), and finally setting bit 0 of control register `CR0`. The final step is sensitive, causing a VM exit to the hypervisor (Figure 2.7, step ①).

On exit, the hypervisor inspects the VMCS's exit information and guest-state areas (step ②), updates internal tracking (e.g., CPU mode), writes updated values back into the VMCS (steps ③ –④), and then re-

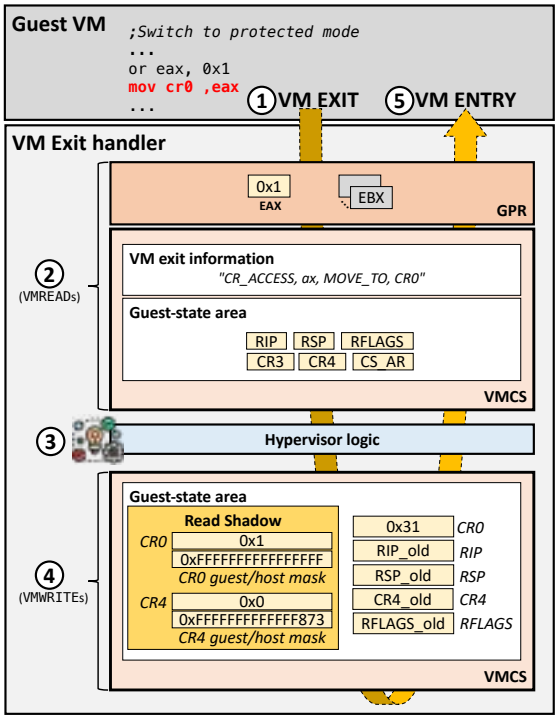


Figure 2.7. Xen hypervisor control flow during switching into protected mode requested by a guest VM [59].

sumes the guest (step ⑤). Even this single operation is realized as a sequence of exits, each an opportunity for subtle flaws.

Real-world vulnerabilities confirm the security risk. Numerous CVEs in Xen and KVM [159] hypervisors (e.g., CVE-2025-21839 , CVE-2024-53135 , CVE-2025-23141) stem from incorrect exit handling, enabling VM escapes or privilege escalation.

Hypervisor Interfaces as an Attack Surface

VM exit handlers are invoked implicitly and at high frequency, often in response to subtle guest state transitions. Their tight coupling with CPU microarchitecture, reliance on extensive state decoding, and interaction with subsystems such as memory management and interrupt delivery make them both performance- and security-critical. As a result, small parsing mistakes, incomplete checks, or ordering flaws can cascade into isolation failures or full VM escapes.

2.1.4 Software Supply Chain Surface

Software Supply Chain. Modern software development increasingly relies on third-party open-source software (OSS) components rather than being built from scratch [80]. Applications are assembled from a vast ecosystem of libraries, frameworks, plugins, and packages maintained by diverse communities and organizations. Collectively, these components form the software supply chain. In all major ecosystems, dependencies are typically obtained from public package registries (e.g., PyPI for Python, npm for JavaScript, Go Modules for Go). This model improves productivity and accelerates development, but it also produces large and complex dependency graphs. Developers not only depend on their direct choices, but also on long transitive chains of code they never explicitly selected. As illustrated in Figure 2.8, even a partial view of the Kubernetes dependency graph shows the magnitude of this problem, where the number and interconnection of dependencies quickly become overwhelming.

Go Ecosystem. GitHub hosts over 1.8 million Golang modules [119]. This vast ecosystem highlights the scale at which open-source third-party code is made available and actively consumed in Go development. Large-scale, industrial Go applications often depend on hundreds of external packages. A notable example is Kubernetes, which currently pulls hundreds direct and indirect Go dependencies to build [221]. In Go, a module [276] is a collection of packages that are released, versioned, and distributed together. Modules may be downloaded directly from version control repositories (commonly Git repositories), or from module proxy servers [275]. Modules are downloaded and built using the standard `go` command-line

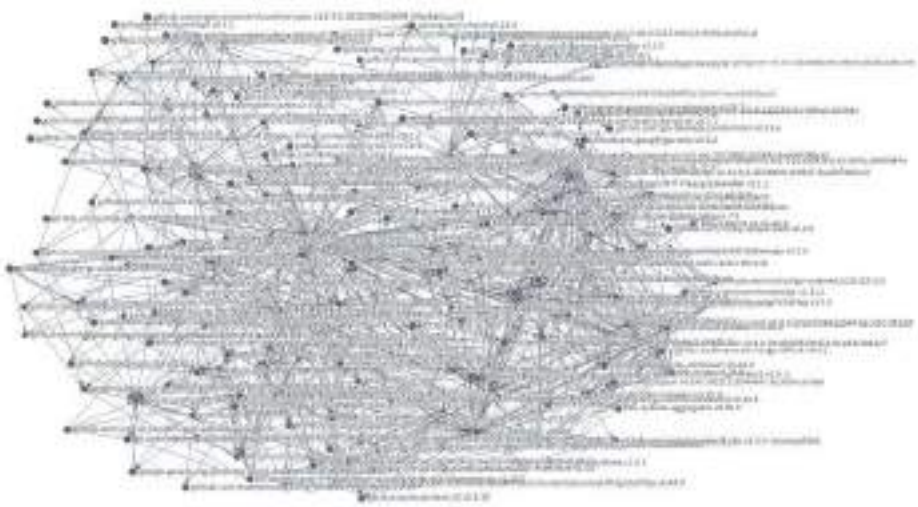


Figure 2.8. Excerpt of the dependency graph of the Kubernetes project (only a portion of the total is shown).

tool. A package refers to a directory containing one or more Go source files in the same namespace and represents the fundamental unit of compilation and encapsulation. A module may consist of one or many packages. A project is a library or an application hosted in a repository with one or several modules. When a Go project is built, all required modules and packages are bundled into a single statically-linked binary. The binary contains the Go runtime [123], which manages initialization, garbage collection, and concurrency. In order to achieve high performance, the Go runtime is designed to be simple and lightweight. The Go module system introduces unique technical challenges. In particular, the activity of packages inside a Go program is opaque from the point of view of monitoring tools. Since the compiled Go program is a flat binary, the OS has no visibility on which Go package is invoking a system call. Moreover, the Go runtime does not provide security features to analyze and manage accesses to platform features and APIs, such as in the Security Manager for the Java language. This lack of observability makes it easier for attackers to slip malicious code into Go programs. The Go module system and toolchain simplify dependency management throughout the phases shown in Figure 2.9, and

described in the following.

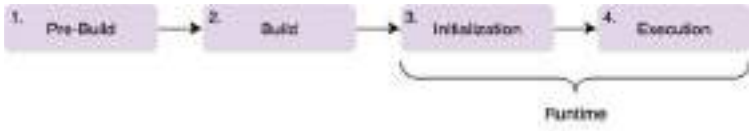


Figure 2.9. Go Package Lifecycle

1. Pre-Build: To create a Go module, developers can use the `'go mod init'` command, which generates a manifest file named `'go.mod'` (shown in Listing 2.1). This file lists all the required direct and indirect dependencies and ensures reproducible builds by pinning dependencies to specific versions [276]. During this phase, dependencies can be explicitly integrated into the Go module through the `'go get'` command, which fetches and caches the required packages locally. After this, developers can test integrated packages (`'go test'`) and utilize code generation tools (`'go generate'`) to automate tasks like creating boilerplate code or generating environment-specific artifacts, before the build phase.

Listing 2.1. Example of a `go.mod` manifest file.

```
1 module github.com/ethereum/go-ethereum
2 go 1.21
3 require (github.com/Azure/.../azblob v1.2.0
4          github.com/Microsoft/go-winio v0.6.1 ...)
```

2. Build: During the compilation of a module, the `'go build'` tool parses the manifest (`'go.mod'`) and automatically fetches any required dependencies not cached locally. The Go compiler then compiles them into machine code [281]. The compiled code is linked with the Go runtime to create an executable binary. Go toolchain is explicitly designed to avoid code execution during this process, ensuring build-time security.

3. Initialization: When executing the installed binary, the Go Runtime first initializes dependencies based on a dependency ordering [277]. If package A imports package B, package B will be initialized before package A. First, package-level *global variables* are initialized, in the order they

are declared. Then, the Go runtime executes any package-level `init()` functions. Global variables initialization and init functions can contain arbitrary code.

4. Execution: Finally, once all packages are initialized, the Go runtime will execute the `'main'` function in the main package, as a *goroutine* [277]. A goroutine is a lightweight thread managed by the Go runtime, allowing for concurrent execution with lighter overhead than with traditional OS threads. The Go runtime also automatically manages memory allocation and garbage collection.

Software Supply Chain Security in Go The software supply chain forms an attack surface not because of a single flaw, but due to its systemic composition. Unlike vulnerabilities in application logic, supply chain risks arise from trust relationships, automated integration workflows, and the absence of fine-grained controls across package boundaries. The core challenge is transitive trust: downstream consumers execute upstream code under the assumption that it is trustworthy and benign. In practice, dependencies often pull in further packages that may be outdated, unmaintained, or even compromised. Dependency graphs, such as the one in Figure 2.8, illustrate the scale of this issue: hundreds of transitive packages, each both a functional component and a potential failure point. In large projects, this accumulation of delegated trust produces latent attack vectors that extend far beyond traditional vulnerabilities in interfaces or logic, amplifying systemic supply chain risk.

A major source of exposure arises from implicit execution paths supported by language features. In Go, constructs such as `init()` functions [278], global variable initialization, and dynamic plugin loading allow code to execute automatically when a program starts, without explicit invocation by the developer. For example, malicious code embedded in a dependency's `init()` function can run before `main()` executes, silently triggering unwanted behavior. Tools like `go generate` [273] further expand automation by inserting code generation steps prior to compilation. While these mechanisms enhance convenience and flexibility, they also create hidden entry points where malicious actions can be initiated during initialization, often outside the developer's awareness.

Another structural issue lies in the monolithic privilege model of com-

piled languages. Once built, an application executes as a single process with unified privileges, regardless of how many libraries or packages it integrates. Go exemplifies this model strongly: its statically linked binaries incorporate all dependencies into one executable, with no runtime sandboxing or separation between packages. As a result, if even a single imported dependency is compromised, for example through typosquatting, repository hijacking, or malicious pull requests, it inherits the full capabilities of the host process. The operating system cannot attribute system calls or sensitive operations to a specific package, and the Go runtime provides no mechanism for package-level isolation. This lack of intra-binary privilege separation means that once malicious code is introduced, it executes with the same capabilities as the rest of the program.

The Go module system itself also introduces risks from the supply chain perspective. Mechanisms for reproducibility, such as module proxies and caching, ensure that builds remain consistent over time even if the original source changes or disappears. While valuable for stability, this persistence can inadvertently preserve malicious artifacts: once a tainted version enters a proxy cache, it may continue to be served to downstream consumers long after the upstream repository has been cleaned or reverted. In this way, caching mechanisms that were designed for integrity and availability can paradoxically amplify compromise longevity.

This systemic fragility is not theoretical, but it was demonstrated in the real-world compromise of the `boltdb/bolt` module [53]. An adversary published a typosquatted package impersonating the legitimate library, preserving normal database functionality while stealthily introducing remote code execution (RCE) capabilities. Specifically, the malicious code established an obfuscated, persistent connection to a command-and-control (C2) server using Go's `net.Dial` API, then executed attacker-supplied shell commands through `os/exec.Command`. This enabled arbitrary file and process manipulation—capabilities never intended by the original BoltDB library. After the malicious version was cached by the Go Module Mirror, the attacker rewrote the Git tag to point to a clean commit, effectively concealing the backdoor in the public repository. Yet the proxy continued to serve the cached pseudo-version, allowing the backdoor to persist in the ecosystem long after the repository appeared clean.

This incident illustrates how transitive trust, caching persistence, and

lack of package-level isolation can combine to produce durable and stealthy supply chain compromises. It also demonstrates why runtime-level observation and enforcement are essential to contain such threats.

To reason about these risks, it is useful to distinguish between two complementary categories of exposure:

- Static exposure: risks introduced by importing packages that expose dangerous or exploitable features, such as reflection, unsafe memory operations, or unbounded network access. These risks are often detectable through static analysis or dependency graph inspection.
- Runtime exposure: behaviors that remain invisible to static analysis, such as conditionally triggered logic, dynamically constructed system calls, or obfuscated payloads. These forms of exposure require runtime tracing, behavioral profiling, or policy enforcement to detect and constrain.

Attackers increasingly exploit runtime exposure using sophisticated obfuscation and evasion techniques. Common tactics include encrypting payloads, leveraging reflection for dynamic invocation, injecting native code via CGO, or embedding inline assembly. Go's runtime features, such as dynamically loaded plugins, external binary execution, and reflection-based invocation, offer flexibility but can equally be abused for persistence and command execution. These mechanisms often bypass traditional static analysis [198] and are best mitigated through runtime enforcement approaches [155]. Unlike conventional exploits that rely on memory corruption or logic flaws, modern supply chain attacks weaponize legitimate language features in unexpected ways, blending seamlessly into normal execution flow.

A distinct class of exploits is represented by repojacking, in which attackers hijack deleted or abandoned repositories by re-registering the original import path. Since Go resolves modules based on their import path, this allows adversaries to distribute malicious packages under names previously associated with trusted projects. Consumers of the original package may remain unaware that the codebase is now controlled by a different, potentially malicious actor.

Taken together, these characteristics, including automatic execution, coarse-grained privileges, long-lived caching, and implicit trust, create a

uniquely broad and stealthy attack surface. The Go software supply chain is not merely a collection of packages; it is a cascade of deeply integrated components whose behaviors are difficult to audit in isolation. As attackers continue to exploit these pathways, securing the software supply chain becomes an urgent priority. Addressing these challenges requires mechanisms capable of observing and constraining package behaviors at runtime, one of the central focus of this thesis.

Software Supply Chain as an Attack Surface

The software supply chain represents an attack surface rooted in software provenance and compositional structure. Unlike traditional attacks that target vulnerabilities in deployed applications, supply chain attacks exploit the trust placed in complex dependency graphs and the propagation of risk through upstream code. In ecosystems like Go, implicit trust in dependencies, auto-executing code paths, and monolithic binaries without package-level isolation allow adversaries to embed malicious behavior within seemingly legitimate packages. Such attacks can persist undetected through caching and runtime evasion techniques. Securing the supply chain therefore requires addressing both static and dynamic exposure, before, during, and after third-party code integration.

2.2 Reducing Attack Surfaces

Reducing the attack surface refers to limiting the number of exposed entry points, minimizing the privileges granted to each, and removing unnecessary functionality. The goal is to shrink the set of interfaces, operations, and resources that can be misused by adversaries, thereby lowering the opportunities for exploitation. In the security literature, attack surface reduction is typically pursued through three complementary approaches: *security hardening*, *security assessment*, and *runtime enforcement*. Each pillar addresses a different state of the system lifecycle, from preventive configuration, to proactive discovery of weaknesses, to runtime containment of residual risks, respectively.

Security Hardening. Security hardening is the process of proactively strengthening a system before deployment to reduce its exposure to attacks. This involves removing or disabling unnecessary services, enforcing strict access controls, applying patches, and adopting secure default configurations. Techniques such as least privilege enforcement, service minimization, and configuration baselining are used to shrink the available attack surface. The primary goal is to minimize exploitable weaknesses before adversaries can take advantage of them.

Security Assessment. A security assessment is the systematic evaluation of a software system to identify vulnerabilities, misconfigurations, and potential risks. It can involve static analysis, such as symbolic execution or taint analysis, which examine code without executing it to detect flaws, and dynamic analysis, such as fuzzing, which feeds unexpected or malformed inputs to running programs to uncover crashes and security issues. Assessments may also include penetration testing and vulnerability scanning. Unlike hardening, which focuses on proactive configuration, assessment emphasizes the discovery and analysis of weaknesses to guide remediation.

Runtime Enforcement. Runtime enforcement techniques defend systems during execution by monitoring their behavior and intervening when suspicious or malicious activity is detected. These approaches dynamically enforce security policies, allowing the system to tolerate residual risks even when vulnerabilities remain. Common techniques include intrusion detection/prevention systems, sandboxing, runtime application self-protection (RASP), and control-flow integrity. By reacting in real time, runtime enforcement helps contain attacks that bypass hardening and escape detection during assessments.

The remainder of this section provides a systematic review of the literature, offering an overview of prior and related works on reducing attack surfaces across each of the five axes we identified.

2.2.1 Hardening of the Orchestration Surface

Research on securing the orchestration layers can be grouped into three main lines: (i) static analysis of configurations, (ii) container-centric

runtime security, and (iii) REST API security. Each research lines contributes valuable protection but also leaves important gaps when applied to orchestration environments.

Static Analysis. Static analysis tools and methodologies have been developed to mitigate security risks issued by misconfigurations, focusing on pre-deployment detection in Kubernetes manifests and Helm Charts. Empirical studies [258, 52, 238] highlight the prevalence of Kubernetes misconfiguration and their security implications. Research analyzing Helm charts and Operators [303, 297] identifies insecure defaults and outdated dependencies that elevate security risks. Broader efforts [257, 208] emphasize adherence to best practices, including RBAC or network segmentation, but these approaches rely on manual intervention and lack runtime adaptability. Static analysis tools, such as KubeLinter [270], Polaris [103], Checkov [237], KICS [64], and SLI-Kube [238] identify misconfigurations using predefined rules. Graph-based methods such as KGSecConfig [132] and others [51, 84] automate secure cluster configuration or provide security assessment of configurations. More recent generative approaches [190, 196, 178] leverage large language models to identify and remediate misconfigurations. RBAC misconfiguration detection tools, such as EPScan [128], focus on identifying excessive access control permissions.

Despite their utility, these tools operate pre-deployment and leave systems exposed to runtime threats, such as malicious API requests that vulnerable unused features exposed through the API. This underscores a central limitation of prior static-analysis work: the absence of runtime, workload-specific enforcement of API policies, which is essential to reduce the attack surface. While we motivate this with Kubernetes, the challenge generalizes to securing complex, highly configurable software systems.

Container Security. Container-centric security solutions primarily address runtime behaviors of containers, targeting process execution and interactions within the host environment. Runtime monitoring tools, such as system call monitoring [158], detect anomalous container behavior but require extensive pre-configuration and rely on heuristic models. ProSPEC [153] predicts potential breaches through proactive policy enforcement. Tools like Kub-Sec [307] and KubeArmor [71] generate security profiles for

pods, enforcing the principle of least privilege. Systems like Sysdig Falco [104] and similar tools monitor container operations based on predefined rules. Mechanisms such as eBPF [95] and Seccomp [96] can also be used to whitelist allowable syscalls from containers.

These solutions effectively constrain low-level container behavior but their scope is limited. The target process activity rather than higher-level APIs remaining not applicable to orchestrator requests. In principle, syscall whitelisting can detect some of the malicious behaviors that may occur after exploiting vulnerabilities in orchestrators, such as executing privileged system calls. However, defining policies for finer-grain syscall filtering is challenging, often requiring complex static or dynamic program analysis, or manual specification. Moreover, certain malicious behaviors may still escape syscall-based policies, for example by abusing privileges already available to an application. This illustrates a general tension in hardening complex software stacks: lower-level interfaces such as syscalls are well-studied, but higher-level abstractions, including rich API surfaces, remain insufficiently protected.

REST API Security. Securing REST APIs has long been a focus in web and application security, with numerous methodologies proposed for generating and enforcing access control policies to mitigate unauthorized access and enhance runtime security. Frameworks like RestPL [187] facilitate precise and flexible policy definitions for RESTful APIs, emphasizing request-level granularity but remaining confined to static pre-deployment configurations. Similarly, Jayathilaka et al. [148] propose a framework for enforcing API security policies in cloud platforms, enforcing best practices during API deployment. Atlidakis et al. [40] extend REST API security through property checking, fuzzing, and runtime monitoring, while a more recent framework by Khan et al. [154] focuses on detecting and mitigating vulnerabilities in REST APIs by integrating reverse proxy techniques to identify and prevent attacks like SQL injection and cross-site scripting in real time.

While effective for traditional REST APIs, these solutions lack mechanisms for dynamically adapting policies to rich configurations and nested specifications of more complex interfaces such as Kubernetes APIs. This gap highlights a broader challenge: although REST API security has ad-

vanced considerably, methods designed for relatively flat request-response models are insufficient when applied to deeply configurable software ecosystems. Strengthening such complex software interfaces requires security hardening approaches that move beyond static analysis and toward adaptive workload-aware enforcement.

Contribution: Hardening of the Orchestration Surface

This dissertation addresses limitations of prior work through the design of *KubeFence* [63], a runtime enforcement system for orchestration-layer APIs. Unlike static analysis tools, *KubeFence* derives workload-specific policies dynamically and enforces them during execution. Unlike container-centric defenses, it targets the orchestration API directly, inspecting the semantics of configuration requests rather than only low-level system calls. Unlike traditional REST API approaches, it supports deeply nested and highly configurable request structures typical of Kubernetes. By providing fine-grained, automated, and runtime-capable enforcement, *KubeFence* reduces the orchestration attack surface in ways that prior approaches cannot, mitigating API-level attacks.

2.2.2 Fuzzing of IPC Surface

Inter-Process Communication (IPC) channels are a privileged yet underscrutinized interface in complex and safety-critical systems. For example, SIPC in MILS deployments for a critical surface yet remain difficult to test in practice. Mitigating this exposure requires systematic, adversarial testing of guards and their IPC interfaces to uncover parser errors, logic flaws, or policy oversights that would otherwise remain latent.

Unlike configuration audits or source-level review, which are often infeasible in closed and proprietary environments, fuzzing is particularly suited to this task: it requires no source code, scales to opaque industrial binaries, and has proved effective at uncovering vulnerabilities in parsers and protocol implementations [192]. Black-box fuzzing, however, lacks the depth needed stateful IPC protocols. Coverage-guided (gray-box) fuzzing is therefore preferable: by tracking execution coverage (i.e., the code blocks that are executes), it steers exploration toward unexplored code

paths, increasing the likelihood of exercising deep parsing logic and policy decisions. While this approach has been effective for general-purpose software, applying it to industrial, closed-source targets like MILS remains challenging, due to three recurring challenges:

(1) Intrusiveness. Fuzzing MILS-based applications requires introducing a *fuzz driver* [146] that submits inputs to the target software. In these systems, applications run inside virtual boards (e.g., the "*guard*" board in a MILS-based gateway) and communicate with other components through SIPC channels in a client-server fashion. To fuzz such applications, a dedicated client board must be added to send fuzzed inputs via SIPC. This involves rewriting, reconfiguring, and rebuilding the target, which may be impractical or infeasible when source code is unavailable (e.g., proprietary binaries). A second difficulty is failure detection. The fuzz driver must identify when the target crashes, yet special-purpose embedded systems often lack visibility into application state. For example, the MILS kernel provides no interface to observe a board's failure from another board. A common workaround is to use liveness checks [107], where the client periodically probes the target. However, MILS-based gateway applications do not produce responses by design. In that case, engineers must introduce an additional SIPC channel for status messages, modifying both client and server architecture. Even then, response-based detection can generate false alarms: delays caused by large fuzz inputs, concurrency, or external events may appear as crashes, leading the fuzz driver to misclassify the system state.

(2) Toolchain Dependency. Even when engineering effort is invested in implementing and setting up fuzzing a fuzz diver and mechanisms for failure detection, industrial embedded systems rarely support the mechanisms needed for coverage-guided fuzzing. Proprietary OSes and compilers generally lack instrumentation features available in open-source toolchains such as LLVM or GCC, and coverage support, when present, is limited to offline reporting (e.g., *gcov* [15]) rather than real-time feedback usable by a fuzzer. As a result, execution coverage remains inaccessible to the fuzz driver and is exposed only to external debugging tools. In the specific case of a MILS-based gateway, the entire system, including kernel, partitions, and applications, must be built, linked, and package with

the proprietary *VxWorks Workbench* toolchain. Workbench neither supports coverage instrumentation options (e.g., `-fsanitize=coverage`) nor provides plug-in hooks like those of Clang or GCC. Consequently, fuzzing must operate in black-box mode. Even with source code available, enabling coverage would require developing custom instrumentation hooks from scratch, tightly bound to that specific toolchain.

(3) Hardware Dependency. Typically, embedded systems do not adopt CPU architectures for general-purpose systems (e.g., x86), but they are based on specialized CPU architectures such as ARM, MIPS, PowerPC and SPARC. For example, the Wind River VxWorks MILS comes with a board support package for the PowerPC architecture. Consequently, this diversity poses a significant challenge for embedded system fuzzing, demanding support for multiple architectures. In addition, these architectures lack advanced hardware features that modern fuzzers use. When compiler-assisted instrumentation cannot be enabled, as in the case for the VxWorks MILS, state-of-the-art fuzzers typically fall back on hardware tracing units (e.g., Intel PT) [66] or debugger-assisted tracing via GDB [98] to recover execution feedback. Unfortunately, MILS-based boards [4] do not offer neither Intel PT nor a GDB Remote Serial Protocol stub, leaving fuzzers without any practical source of execution traces.

These recurring challenges motivate five requirements for fuzzing industrial IPC mechanisms: *R1 Binary-only*, *R2 Non-intrusive*, *R3 Toolchain independence*, *R4 Hardware independence*, and *R5 Full-system scope.* In the remainder of this section we review the main lines of prior work and highlight their limitations in meeting the requirements of fuzzing industrial embedded systems.

Static Binary Rewriting. Fuzzing techniques such as AFL-Dyninst [9], RetroWrite [87], and AflIoT [92] instrument binaries without requiring source code by analyzing and modifying their compiled binary. Using frameworks like Dyninst, they inject instrumentation directly into the binary, enabling coverage-guided fuzzing. However, current rewriters are implemented only for mainstream ISAs such as x86 and ARM64. This leaves PowerPC, SPARC, and other architectures typical of industrial platforms unsupported, thereby falling short on *R4 (Hardware independence).* In

addition, static rewriting has so far proved practical only for user-space programs. Extending it to kernels, hypervisors, or the multi-partition layout of a MILS system remains an open challenge, leaving *R5 (Full-system scope)* unmet.

Dynamic Binary Instrumentation. DBI-based fuzzers such as AFL-DynamoRIO [8] and AFL-PIN [11] insert probes at runtime via frameworks like DynamoRIO and Intel PIN. Because probes are injected into already-compiled executables, these techniques work on opaque binaries without compile-time support, satisfying the toolchain independence. Yet portability is bounded by the underlying DBI framework, and support for industrial ISAs remains absent leaving *R4 (Hardware independence)* unmeet. Moreover, mainstream DBI frameworks operate only at user privilege level, excluding kernel and hypervisor code, and therefore fail *R5 (Full-system scope)*.

Hardware-Assisted Techniques. Hardware-assisted fuzzing techniques such as Ptrix [66], Honggfuzz [6], kAFL [253], μAFL [182], and GDB-Fuzz [98] leverage hardware-assisted tracing capabilities such as Intel PT or debugging interfaces, to collect coverage information. These approaches achieve high precision on opaque binaries without intrusive modifications, but once again depend on x86-specific tracing features, leaving *R4 (Hardware independence)* unmet for MILS deployments. Only a subset (e.g., GDBFuzz and kAFL) extend tracing into kernel mode, so *R5 (Full-system scope)* is only partially addressed. AflIoT [92] similarly adopts a hardware-dependent approach by fuzzing directly on IoT devices rather than relying on emulation, enabling peripheral interaction but restricting applicability to devices that support its instrumentation method. This line of work shows how hardware support can accelerate fuzzing, but at the cost of generality and portability.

Emulation-Based Approaches. User-space fuzzers, such as AFL's QEMU mode [7], AFL++ [109], and UnicoreFuzz [188] rely on QEMU user-mode emulation to run application binaries (e.g., ELF executables) on a host system. Instructions are translated to the host ISA, while system calls are intercepted and executed by the host kernel. The introspection

capabilities of the emulation layer enables dynamic binary instrumentation of user-space programs, supporting gray-box fuzzing without source code access or compile-time instrumentation. However, these tools cannot address *R5 (Full-system scope)*. QEMU user-mode emulation is designed for standalone applications running atop a general-purpose OS, not for monolithic embedded system images. For instance, the VxWorks MILS gateway integrates the RTOS, hypervisor, partitions, and applications into a single binary that expects to run on bare metal, which user-mode emulation cannot execute. Even when industrial software is shipped as application binaries, they often rely on proprietary RTOS interfaces with non-Linux syscall ABIs or custom loaders, further breaking compatibility with QEMU user-mode.

Fuzzing for Specialized Targets. Specialized fuzzers such as Syzkaller [5], Tardis [259], and AflIoT [92] target specific domains like kernels, embedded software, or IoT binaries. Syzkaller and Tardis achieve deep kernel coverage by relying on custom fuzz drivers and source-level instrumentation, but this dependence breaks *R1 (Binary-only)*, *R2 (Non-Intrusive)*, and *R3 (Toolchain independence)*. They also restrict their scope to the kernel, leaving user partitions and hypervisors untested, and thus fail *R5 (Full-system scope)*. AflIoT, by contrast, applies static binary instrumentation directly on Linux-based IoT devices, enabling on-device fuzzing without emulation, but remains tied to platforms that support its instrumentation method, again violating R4 and R5. Overall, these tools demonstrate that domain-specific fuzzers can be highly effective in their niche, yet they lack the generality required for undocumented, multi-layered, and proprietary industrial software stacks.

Table 2.1 summarizes the limitations of state-of-the-art binary fuzzing tools when applied to specialized industrial systems, such as MILS-based applications. The main gap is their inability to provide full-system coverage across user-space libraries, kernels, and hypervisors.

Table 2.1. Limitations of the State-of-the-Art tools for binary-level fuzzing.

Technique	Binary Fuzzing	Non-Intrusive	Tool Independ.	Hw Independ.	Full-System
AFL-Dyninst [9]	✓	✓	✓	✗	✗
RetroWrite [87]	✓	✓	✓	✗	✗
AflIoT [92]	✓	✗	✓	✗	✗
AFL-DynamoRIO [8]	✓	✓	✓	✗	✗
AFL-PIN [11]	✓	✓	✓	✗	✗
Ptrix [66]	✓	✓	✓	✗	✗
Honggfuzz [6]	✓	✓	✓	✗	✗
kAFL [253]	✓	✗	✓	✗	✓
μAFL [182]	✓	✓	✓	✗	✗
GDBFuzz [98]	✓	✓	✓	✗	✓
AFL QEMU mode [7]	✓	✓	✓	✓	✗
AFL++ QEMU mode [109]	✓	✓	✓	✓	✗
UnicoreFuzz [188]	✓	✓	✓	✓	✗
Syzkaller [5]	✗	✗	✗	✓	✗
Tardis [259]	✗	✗	✗	✓	✗

Contribution: Fuzzing of the IPC Surface

This dissertation addresses the limitations of prior fuzzing approaches by introducing *FuzzBox* [62], an emulation-based system fuzzer for industrial, binary-only targets. *FuzzBox* requires no source code, and no architectural modifications to insert fuzz drivers or liveness checks, satisfying R1 (Binary-only) and R2 (Non-intrusive). By avoiding compiler dependencies or specific toolchains, it achieves R3 (Toolchain independence). It operates without specialized tracing units or debug stubs, supporting diverse CPU architectures beyond commodity x86, thus meeting R4 (Hardware independence). Finally, by relying on full-system QEMU emulation rather than user-mode execution, it runs unmodified system images, including kernels, hypervisors, and IPC channels, covering deeply stateful components and achieving R5 (Full-system scope). In this way, *FuzzBox* enables practical gray-box fuzzing of closed-source industrial software, from MILS guards to embedded firmware in industrial machinery and IoT device.

2.2.3 Fuzzing of Hypervisor Surface

The hypervisor layer represents one of the most privileged and security-critical components in modern virtualization stacks. Its interfaces, including VM exits, hypercalls, and I/O traps, are directly influenced by guest-controlled state and execute in highly privileged contexts. This makes them a prime target for attackers seeking to trigger isolation failures or VM escapes. A range of methodologies have been proposed to test and validate hypervisor security. Traditional approaches include *formal verification* [161, 112, 181, 56, 47, 138], which provides strong guarantees of correctness but is extremely costly and difficult to scale across the entire hypervisor code base and due to the complexity of modeling hardware-assisted virtualization interfaces. Other methods, such as *manual auditing* [193, 78] or *fault injection* [129, 152, 57], provide partial coverage but remain heavily dependent on expert knowledge. While these approaches offer useful insights, their practical deployment at scale is limited, and they often fail to exercise the dynamic, state-dependent behaviors of real-world hypervisor interfaces. Thereby, fuzz testing has emerged as a practical and scalable strategy to reduce the hypervisor attack surface by stress-testing privileged interfaces under diverse guest-driven conditions. Fuzzing aims to expose isolation failures, logic bugs, and state-dependent corner cases by executing realistic interaction sequences at high throughput [308]. Prior research on hypervisor fuzzing can be grouped into three main lines: CPU virtualization testing, device virtualization testing, and record-and-replay techniques.

CPU Virtualization testing. *Amit et al.* [32] propose to apply the testing environment of CPU vendors to hypervisors, however, this approach need strong awareness of x86 architecture to generate comprehensive test cases. *PokeEMU* [299] generates CPU test cases for virtual CPU implementations by applying symbolic execution to executable specifications, but it primarily targets hypervisors without hardware-assisted virtualization. *MultiNyx* [111] focuses on hardware-assisted virtualization using dynamic symbolic execution, but incurs significant performance overhead by recording multiple traces between VM and VMM contexts. *HyperFuzzer* [115] combines fuzzing with commodity hardware tracing, avoiding full hypervisor execution tracking but still relying on instrumentation.

While these studies highlight the feasibility of fuzzing CPU virtualization paths, they generally require expert knowledge to construct initial fuzzing seeds manually, and they often neglect I/O device virtualization behaviors, limiting their coverage of real-world hypervisor interfaces.

Device Virtualization testing. Device virtualization introduces additional complexity, as the hypervisor's behavior depends on the VM's architectural state during device emulation. *Schumilio et al.* [251] discovered available hypervisor interfaces using a custom OS and tested them with a black box fuzzer accelerated by a bytecode interpreter, but the fuzzing seeds wre against manually built. *Nyx* [252] applied nested virtualization and grammar rules to specify the structure of the target emulated devices, but required manually defined input grammars. Subsequent work introduced record-and-replay of guest-device interactions. For example, Henderson et al.[139] replayed MMIO activity of virtual devices in QEMU, while VShuttle[230], Morphuzz [55], and MundoFuzz [200] fuzzed broader device interfaces including DMA. VShuttle and Morphuzz required hypervisor instrumentation of DMA APIs, whereas MundoFuzz collected I/O instructions and DMA operations inside the guest OS without modifying the hypervisor. MundoFuzz also applied grammar inference and statistical learning to derive input semantics, but remained sensitive to interleave asynchronous events (e.g., timer interrupts) that generate coverage noise. While these device-focused studies advanced coverage of emulated components, they often relied on manual input grammars or heavy instrumentation, limiting scalability across diverse hypervisor implementations. Moreover, device interfaces are comparatively well-tested within hypervisors, whereas equally critical components such as CPU virtualization remain far less explored and thus insufficiently covered by this line of work.

Record and replay. Record and replay (RnR) techniques are widely used in fuzzing to capture and reproduce valid interaction sequences, facilitating grammar inference and test case generation [200, 139, 230, 260]. Beyond fuzzing, RnR is also used in security to analyze and debug execution traces. In these cases, recorded events are injected at precise times to enforce deterministic replay. For example, replay has been used to verify suspected exploits and to distinguish true positives from false posit-

ives [256]. RnR is likewise employed to analyze time-of-check-time-to-use (TOCTOU) race conditions [93] and to determine whether systems were compromised after the discovery of zero-day vulnerabilities [151]. While effective for debugging and forensics analysis, applying RnR directly to fuzzing complex hypervisor interfaces remains challenging due to performance overhead and the difficulty of capturing valid VM states.

Prior work leaves key gaps: CPU fuzzers need manual seeds and miss device behaviors, device fuzzers rely on grammars or instrumentation and neglect CPU paths, and RnR incurs high overhead or fails to capture realistic guest-driven states. None combine low-overhead replay with faithful VM state to systematically drive hypervisors into comple, hard-to-reach execution paths.

Contribution: Fuzzing the Hypervisor Surface

This dissertation introduces *IRIS*, a hypervisor fuzzing framework that enables seed acquisition and replay to *preserve realistic VM and hypervisor states* from actual guest execution. Replay in *IRIS* is not limited to reproduction but is used to deterministically drive the hypervisor into complex states before applying mutations, allowing the exploration of deep execution paths and the discovery of otherwise unreachable bugs. Unlike prior CPU virtualization fuzzers, *IRIS* avoids manual seed engineering and covers device interactions. Unlike device fuzzers, it covers also CPU virtualization components in the hypervisor. Unlike existing record-and-replay approaches, it achieves fidelity without prohibitive overhead by capturing and replaying guest-driven state transitions. A prototype demonstrates accurate coverage reproduction and significant efficiency gains, providing a practical route to exercise under-tested hardware-assisted virtualization paths.

2.2.4 Securing the Software Supply Chain

Modern software development relies on vast ecosystems of third-party components. Each imported dependency broadens the attack surface of the integrating software system, creating opportunities for adversaries to inject, propagate, or activate malicious code. We refer to this exposure as

the *software supply chain attack surface* (see Section 2.1.4). Unlike traditional vulnerabilities, supply chain attacks exploit the *composition process*, targeting everything from code acquisition and transitive dependency trees to build pipelines and runtime extensions (e.g., dynamically loaded plugins, foreign-function interfaces). Mitigating this attack surface requires approaches that span the full lifecycle of software. Research in this area can be grouped into three lines of work: (i) *taxonomies*, which classify and characterize attacks and attack vectors; (ii) *static analysis*, which analyze dependencies before deployment; and (iii) *dynamic analysis*, which monitor or constrain behavior at runtime. While most research targets ecosystems like JavaScript, Python, or Java, Go warrants attention. This language has been heavily used for critical cloud software [293], including Kubernetes [168], Docker [89], Terraform [134], Etcd [100], and several others mission-critical infrastructure [85]. Injecting malicious code into these Golang software projects would enable unauthorized control over cloud infrastructures, with potentially devastating consequences. Thus the security of the Go ecosystem has systemic impact, yet Go-specific supply chain studies remain limited. We therefore review existing work with an emphasis on its applicability to Go.

Supply Chain Attack Taxonomies. Taxonomies aim to categorize and classify threats, establishing comprehensive knowledge for software supply chains. Ohm et al. [218] collect and analyze 174 malicious real-world packages exploited in supply chain attacks for JavaScript, Java, Python, PHP, and Ruby packages. The analysis grounds two taxonomies. The first taxonomy classifies 18 attack methods for injecting malicious code into a dependency tree. The second one classifies 8 vectors for executing the injected malicious code, belonging to one of two branches: a software lifecycle phase (install scripts, test case, and runtime), or a conditional trigger. Ladisa et al. [176] extend the Ohm et al. taxonomy for malicious code *injection* to 107 attack vectors. They survey gray and white literature, considering exploited and non-exploited attacks, but do not categorize execution strategies for malicious code after injection. A complementary taxonomy is proposed in Hitchhiker [177], which considers language and package manager features exploitable for executing malicious code, identifying seven execution techniques spanning install-time,

and runtime across seven languages, including Go. Collectively, these taxonomies underscore the diversity of supply chain attacks. However, they remain largely ecosystem-agnostic, without considering Go's specific language or package manager features that can be exploited during an attack. As a result, they provide only general classifications, not concrete attacker methods tailored to Go.

Contribution: Go-Specific Supply Chain Taxonomy

This dissertation introduces the first taxonomy of software supply chain attack vectors specialized for the Go ecosystem. Unlike prior ecosystem-agnostic classifications [218, 176, 177], this taxonomy is grounded in the Go package lifecycle and identifies 12 execution vectors across pre-built, initialization, and runtime phases. By mapping Go-specific language features and package manager behaviors to concrete attacker methods, it provides a systematic framework for reasoning about the attack surface of Go modules. This conceptual contribution establishes the foundation for automated analysis and enforcement of supply chain defenses in Go.

Static Analysis. Static analysis approaches aim to identify threats in dependencies before deployment. Existing work can be grouped into three categories.

Vulnerability Scanners. A first line of work detects known vulnerabilities in imported open source code. Latent vulnerabilities in the imported code could be a powerful mean for supply chain attacks, with attackers targeting dependencies deeply nested in the graphs as entry points for compromise. Tools like *Govulncheck* [226] and *Nancy* [264] adopt a database-driven approach for vulnerability detection. Govulncheck leverages the Go Vulnerability Databases [121], while Nancy is powered by Sonatype OSS Index [265]. While effective at catching previously catalogued issues, these tools cannot identify novel attack surfaces or capabilities introduced through new code. Their coverage is inherently limited to known CVEs.

Code Quality and Capability Analyzers. Other tools inspect source code of Go dependencies for insecure constructs. *Go vet* [224] reports code quality and correctness issues such as unused variables and potential race conditions. *Gosec* [225] applies a rule-based approach to detect 36

classes of security issues, such as URL injection and path traversal. *Go AST* [227] enables user-defined checks via the *Rego* policy engine, though its effectiveness is limited for complex constructs such as native code linking or misuse of language's features. *Capslock* [124] takes a distinct approach, analyzing call graphs derived from AST and SSA representations to identify privileged operations, including network and filesystem access, reflection, CGO, plugins, and external execution. While this provides valuable insights into a package's potential attack surface, Capslock remains limited to capability enumeration and lacks systematic coverage of code execution vectors. It cannot detect new classes of attack behaviors or dynamically triggered threats.

Malicious Pattern Scanners. A significant body of research focuses on mitigating software supply chain attacks by scanning for malicious patterns or anomalies through static analysis. These approaches target malicious package updates, installation-time attacks, and privilege escalations. Latch [295] traces system calls during installation to generate permission manifests, preventing preventing packages from performing unexpected actions in install scripts. Ohm et al. [219] propose a differential capability analysis approach that compares new package versions against trusted baselines and restricts capabilities through a modified Node.js runtime. iHunter [184] performs static taint analysis on iOS SDKs to detect privacy violations such as cross-library data harvesting, using symbolic execution and NLP-assisted API modeling. Other tools such as GuardDog and Amalfi [82, 254], focus on ecosystem-level vetting. They detect suspicious packages at publication time by flagging known malware patterns, anomalous metadata, or typosquatting attempts. While effective for predeployment vetting and filtering obviously malicious packages, these approaches remain offline and cannot capture behaviors that adversaries introduce through obfuscation or delayed execution, which only manifest at runtime.

Contribution: Static Analysis of Go Attack Vectors

Building on the proposed taxonomy, this dissertation develops *GoSurf*, a static analysis tool that detects Go-specific attack vectors in codebases. *GoSurf* leverages AST and SSA representations to analyze package call graphs and identify privileged operations such as reflection, CGO plugins, and external execution. Unlike vulnerability scanners [226, 264], which are restricted to known issues, or linters and capability checkers [224, 225, 227, 124], which provide coarse-grained or rule-based checks, *GoSurf* systematically reports potential execution vectors introduced by dependencies. In this way, it makes the taxonomy actionable, enabling developers to assess and reduce the attack surface of Go software before deployment.

Dynamic Analysis. Dynamic approaches enforce security during execution and fall into two categories.

Application-Level Sandboxing. A number of systems confine or restrict application behavior through runtime sandboxing. NatiSand [26] and Cage4Deno [25] use Linux security features (e.g., Seccomp, eBPF, Landlock) to restrict native extension in JavaScript runtimes like Node.js and Deno. NatiSand relies on manual JSON-based configuration to restrict operations such as file or network access, while Cage4Deno adds support for isolating subprocesses, but lack awareness of finer-grained entities such as npm packages. Similarly, HODOR [291] shrinks the attack surface of Node.js applications by auto-generating syscall whitelists. While it achieves thread-level granularity, it cannot distinguish which dependency initiated a syscall, limiting its ability to isolate malicious libraries. Other approaches extend sandboxing to container-level hardening. Confine [246] statically generates restrictive Seccomp syscall policies for containers via whole-binary analysis, but lacks intra-container granularity or obfuscation resilience. μPolicyCraft [50] synthesizes stateful syscall automata for microservices through symbolic execution and enforces them via a telemetry monitor, yet still operats at the microservice/container level rather than inside processes. Finally, μSCOPE [245] dynamically traces memory accesses and privilege operations in lage codebases (e.g., the Linux kernel) to identify overprivileged regions. While effective for analysis and

privilege planning, it does not enforce runtime behavior and lacks syscall-level or per-package enforcement. Overall, application-wide sandboxing approaches effectively reduce risk by confining large execution environments. However, they operate at a coarse granularity, treating entire applications or containers as a single unit, and therefore cannot restrict or isolate specific malicious libraries within those environments.

Package-Level Permission and Isolation Systems. Another line of work enforces security policies at the level of packages or libraries. MIR [290] uses static analysis to assign read-write-execute capabilities to Node.js libraries, applying them at runtime. ZTD$_{\text{JAVA}}$ [33] similarly enforces package-level permissions in Java via a combination of manual configuration and runtime monitoring. Both approaches help limit vulnerabilities in dependencies but do not account for maliciously added capabilities and lack support for obfuscated behavior. Other systems rely on manually specified permissions or operate over high-level, language-specific API. Ferreira et al. [108] propose a Node.js permission system configured by developers to constrain packages at the JavaScript API level, with no syscall visibility. FLEXDROID [255] enforces privilege separation in Android apps by intercepting Dalvik calls, but relies on Android permission model and does not observe OS-level behavior. Stronger isolation has been explored via system call interposition and virtualization techniques. Codejail [294] transparently confines dynamically linked libraries in Linux via a dual-process model that mediates memory and syscall behavior with the main program. LibVM [126] builds virtualized execution environments for shared libraries, with hardware-assisted or ptrace-based syscall control. Enclosure [117] similarly isolates untrusted libraries by associating closures with restricted memory views and syscall filters, enforced via Intel VT-x. It supports package-level policies in Go and Python, but requires developers to annotate code explicitly and integrate language-level constructs. While all three systems support native code and intra-process isolation, they lack automated policy generation, and require developer effort or manual integration into host applications. In summary, package-level permission and isolation systems move toward finer granularity than application-level sandboxes, enabling enforcement at the level of individual libraries. However, their reliance on manual policy definition or intrusive code annotations, limits scalability.

Contribution: Runtime Enforcement of Package Capabilities

This dissertation introduces *GoLeash*, a runtime system that enforces syscall-level capabilities at the *package* granularity for Go binaries. In contrast to coarse application- or container-level sandboxes [26, 25, 291, 246, 50, 245] and library isolation mechanisms requiring manual policies or intrusive integration [290, 33, 108, 255, 294, 126, 117], *GoLeash* (i) requires no source changes, (ii) provides fine-grained syscall-level control scoped to Go packages, (iii) automates policy generation, and (iv) detects and constrains obfuscated or delayed-execution behaviors at runtime. This package-aware enforcement model directly targets malicious capabilities introduced via supply chain compromise within Go software, and that exhibit their behaviors at runtime.

Chapter 3

Hardening of Orchestration Surface

3.1 Overview

Modern orchestration systems such as Kubernetes (K8s) expose highly privileged management APIs that enable large-scale automation but also expand the attack surface. Misconfigured or overprivileged API calls have been repeatedly exploited in recent years for cryptojacking [49] and data exfiltration [300] campaigns. Although security-by-design principles [289, 288, 102] advocate for minimizing exposed functionality and enforcing least privilege [249, 262], applying these principles to K8s remains difficult. The K8s Role-Based Access Control (RBAC) system governs permissions at the resource level (e.g., Pod, Service, Deployment), but not within their nested YAML specifications, where single fields can unlock privileged features such as host access or device control. As a result, a malicious or compromised client can exploit these overexposed fields even when RBAC policies appear secure.

To address this gap, this chapter introduces *KubeFence*[1], a framework for fine-grained control of the Kubernetes attack surface. *KubeFence* analyzes applications from trusted repositories (e.g., Kubernetes Operators) and automatically generates restrictive security policies that limit API access to the exact resources and specification fields required by each ap-

[1] Available as open-source at: `https://github.com/dessertlab/kubefence/`

plication. In doing so, it enforces least privilege not only at the resource level but also within complex API payloads, effectively extending RBAC with content-aware hardening.

We evaluate *KubeFence* using a catalog of malicious specifications derived from known CVEs and configuration vulnerabilities. Across several widely used Kubernetes applications, *KubeFence* successfully mitigates all tested exploits and misconfigurations, reducing the effective API attack surface by 35% compared to standard RBAC policies. The resulting performance overhead, approximately 50 ms (21%) for cluster management operations, is acceptable given that enforcement occurs only during deployment and not workload execution.

3.2 Attack Surface across Workloads

We have seen that the K8s API is exposed to attacks against K8 misconfigurations and vulnerabilities. We hypothesize that, in practice, many of these vulnerabilities and misconfigurations can be triggered by exploiting specific features of the K8s API that are not required by many users. If this hypothesis holds, it would be possible to prevent attacks by blocking unnecessary features in a workload-specific manner, thus reducing the attack surface of K8s.

To test this hypothesis, we analyzed past vulnerabilities in K8s, and which features can trigger them from the API. We first analyzed the official K8s Vulnerability Database [165], covering all entries from July 2016 to December 2023. This effort yielded a total of 49 CVEs with CVSS scores ranging from 2.6 (low severity) to 9.8 (high criticality). By examining the source code files modified by the patches for these CVEs, we were able to map each vulnerability to the corresponding affected K8s components. These components span a wide range of functionalities, including admissions controllers, kubelet, API server, etcd, kubectl, scheduler, networking, storage, the legacy cloud provides support and security features. We provide the full mapping in the project repository.

Then, we adopted a set of *workloads* to exercise the features exposed by the K8s API. We chose K8s end-to-end (e2e) tests for this purpose [171]. e2e tests were selected because they perform realistic interactions with the K8s API, by deploying resources and managing configurations, as

Figure 3.1. Number of e2e tests in each category that interact with vulnerable files associated with a CVE [63].

seen in production environments. Unlike simple resource operations (e.g., create, delete), these workloads involve complex API requests that selectively trigger different K8s features. For example, an e2e test that manages CustomResourceDefinitions (CRDs) [170] uses service names, ports, and conversion strategies, which exercise specific parts of the K8s codebase. This makes e2e tests ideal as workloads to analyze the relationship between vulnerabilities and features of the K8s API. If a vulnerability can only be triggered through a specific feature, we expect to see that only a small minority of tests cover the vulnerable code.

We selected all available e2e tests across 12 different categories (e.g., networking, storage, scheduling, autoscaling, etc.), excluding tests designed for *Windows* environments because our testbed is built on Linux, and tests in the *disruptive* category, as their focus is on resilience and fault tolerance rather than functional testing. In total, we selected $6,580$ e2e tests. It is important to note that the distribution of e2e tests is not uniform across K8s components. This depends on the richness of configuration parameters available for some components (e.g., storage), where the higher number of tests reflects a greater variety of associated workloads. Therefore, we chose not to sample the tests, and to include the full test suites. We need to consider this imbalance when interpreting the results. Before execution, we instrumented the codebase to collect code coverage data, allowing us to track which lines of source code were accessed by each test [114]. We cross-reference these data with the vulnerable files identified earlier.

Figure 3.1 illustrates the total number of e2e tests grouped by category (columns of the matrix), where each test category interacts with different

parts of the K8s API and requires different K8s resources. In addition, the figure shows, as a heatmap, the number of tests that cover vulnerable code linked to 3 CVEs (rows of the matrix). We found that vulnerable code is covered only by a very small minority of workloads. For example, in the case of CVE-2023-2431, only two workloads from the storage e2e tests cover the vulnerable code. The vulnerable code for the other 46 CVEs is not covered by any of the tests, and not shown in the figure for the sake of brevity. In total, only 29 out of 6,580 tests (i.e., less than 0.5%) exercised a vulnerable part of the codebase. Even if the distribution of e2e test across categories is skewed towards storage, which accounts for the majority of tests, the skew does not undermine this conclusion. We find that, even when excluding the largest category, vulnerable code is covered by only 21 out of 960 tests (i.e., around 2%).

In conclusion, we investigate the overlap between the features commonly used by workloads and the ones exploitable by attacks. Since the overlap is small in practice, disabling unnecessary features when executing a particular workload can thwart many attacks. This preliminary analysis shows that by enforcing API access controls tailored to specific workloads, we can significantly limit exposure to (potentially vulnerable) components that are not necessary, thereby reducing the risk of exploitation. This workload-specific filtering approach can thus minimize the K8s attack surface, addressing a critical gap in the existing coarse-grained RBAC model.

3.3 Threat Model

Our threat model assumes attackers who have gained control over the cluster and can execute commands against the K8s API. These attackers include compromised users with stolen credentials, over-privileged users, and other types of insider threats [74]. Such attackers may attempt to escalate privileges to gain full control over K8s resources (e.g., Pods, Deployments, Services), access the underlying hosts in the cluster, and disrupt provided services. To achieve these objectives, they can misuse the API to deploy malicious resources or reconfigure existing ones with harmful *specifications*, in order to leverage cluster misconfigurations or exploit vulnerabilities in the K8s codebase. An example of attack based on this threat model was described in Section 2.1.1.

Other security threats, such as physical attacks on infrastructure (e.g., compromising the physical machine hosting the etcd database) and supply chain attacks (e.g., targeting container images or CI/CD pipelines) are considered out of scope for this work. In addition, we do not consider volumetric denial-of-service attacks, such as API flooding with high-volume request patterns. Our threat model still considers "non-volumetric" DoS that may be caused by malicious API requests from CVE exploits, which can disrupt workloads and cause service unavailability.

To sum up, we discussed how the existing K8s security model based on RBAC is insufficient to prevent abuses of cluster misconfigurations and exploitation of vulnerabilities. The analysis of K8s vulnerabilities showed that they can be exploited only through specific features of the K8s API. Therefore, we design *KubeFence* for fine-grain filtering of K8s API requests to reduce the K8s attack surface.

3.4 Challenges

Our *KubeFence* solution is based on the fundamental idea of leveraging K8s Operators to obtain strict security policies. K8s operators are becoming more and more popular, and represent a paradigm shift to manage clusters. With operators, developers implicitly encode choices on which features they use. However, the current security model of K8s does not leverage this as an opportunity to restrict the large attack surface. There are technical challenges in doing so.

KubeFence generates security policies from Operator configurations, commonly defined using Helm Charts. These charts bring flexibility through templating, which includes conditionals, loops, data types, and user-defined overrides (Sec. 2.1.1). The challenge lies in ensuring that security policies generalize across all valid configurations that can be derived from charts. To address this, *KubeFence* systematically explores the configuration space of an Operator, by identifying valid variants of the configuration, while restricting them to specific attributes and values where possible.

In addition, K8s API requests contain deeply nested objects with flexible, optional fields, making precise validation challenging. Traditional RBAC only checks K8s resources and actions, whereas fine-grained enforce-

ment requires inspecting the full request structure. A flat-object approach would overlook dependencies between nested fields, enabling attackers to bypass restrictions. To address this, *KubeFence* employs a tree-based validation mechanism that mirrors the hierarchical structure of Kubernetes API requests.

Finally, the architecture of *KubeFence* should fit between clients and the K8s cluster, by ensuring that API requests cannot bypass security validation, with minimal resource and performance overheads.

3.5 KubeFence Design

KubeFence is a proxy-based enforcement mechanism designed to automatically generate and enforce fine-grained API security policies, tailored to specific K8s workloads. A *security policy*, in our context, defines allowable resource specifications, which restricts the attack surface by filtering API requests that include unnecessary attributes. These policies are represented as *validators*, a machine-readable format used by our proxy to check API requests. The proposed approach is tailored for Helm-based K8s operators, which are widely used to manage complex Kubernetes configurations and require careful security analysis [303].

Unlike static security policy checkers that only define allowed resource specifications, *KubeFence* enforces these policies dynamically at runtime. By intercepting and validating every API request during cluster operations, *KubeFence* ensures that only authorized configurations are applied, effectively preventing unauthorized API interactions that could bypass static security measures.

KubeFence seamlessly integrates into the existing Kubernetes ecosystem with minimal disruption to the workflow of administrators and developers. Its operation involves three primary steps, as depicted in Figure 3.2: (1) analyzing K8s workloads and their Helm charts to identify required K8s objects and enumerate their potential configurations; (2) generating security policies based on this analysis and configuring an API proxy to enforce them; (3) intercepting incoming API requests, validating them against the defined policies, and blocking any that deviate from the allowed configurations. By automating policy generation and enforcement, *KubeFence* reduces the manual effort required to secure Kubernetes de-

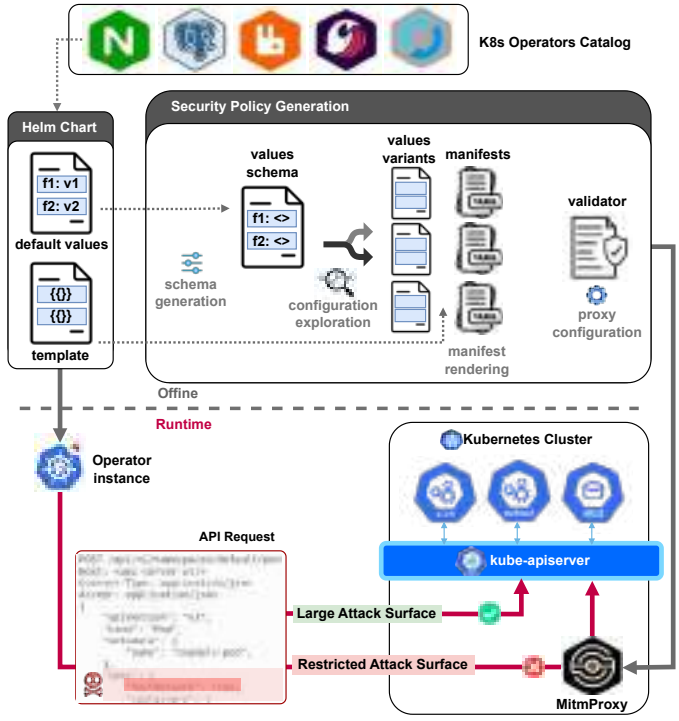


Figure 3.2. KubeFence overview [63].

ployments while enhancing fine-grained protection against insider threats.

3.5.1 Generation of Security Policies

Writing security policies is a complex and error-prone process, especially for complex systems such as K8s. Inaccurate or incomplete policy definitions can leave clusters vulnerable to overly permissive access control.

To address this challenge, *KubeFence* automates the generation of fine-grained security policies by analyzing Helm charts (default values and templates) as input. The goal of *KubeFence* is to produce a consolidated policy in the form of a *validator*, that is, a reference schema for the validation of

incoming API requests. *KubeFence* ensures that the K8s object specification in an API request, i.e., a manifest (see also Figure 2.2), complies with the fields and values of the schema. The schema defines all allowable configurations for each K8s resource defined by the Helm chart. An API request that uses an attribute not included in the schema can be blocked, since it is unnecessary according to the Helm chart. Similarly, if the schema assigns a specification attribute with a fixed value, or a value taken from a small set, API requests that use any value outside this defined range can be blocked.

However, there are several aspects that need to be handled for accurate security policies. (1) Conditional logics in Helm charts dynamically vary the structure of the specification based on user-defined values (as in Figure 2.3). This variability means that many potential configurations generated by these conditions need to be accounted for in the policy. (2) Moreover, while Helm charts often provide default values that fix the structure and content of manifests, users can still override these defaults with custom values (as illustrated in Figure 2.2). Thus, policies should not rely solely on the templates of the Helm charts but should account for such overrides from the user of the K8s operator. (3) Finally, the K8s specification includes critical attributes that are recommended by security best practices (such as `runAsNonRoot`) but that may be omitted in the Helm charts. Policies must ensure these critical attributes are enforced in API requests, regardless of their presence in the Helm charts.

These aspects make policy generation more nuanced. *KubeFence* manages them by exploring the configuration space represented in the Helm charts, to ensure that policies cover all legitimate API requests, while guarding against potential API misuses. The policy generation process is divided into four phases, detailed below.

Generation of values schema. This phase analyzes values of K8s resource specifications in the Helm charts, in order to identify the domain of every field. The value schema will serve as a basis for exploring the configuration space in the next phase.

KubeFence performs a transformation of the default values to produce a structured *values schema*. This transformation aims to: (1) Replace static values with placeholders representing data types or valid ranges, such as

```
# Default Values File          |# Values Schema
image:                         |image:
 registry: docker.io           |  registry: docker.io
 repository: bitnami/mlflow    |  repository: bitnami/mlflow
 pullSecrets:                  |  pullSecrets: [list]
    - name: secret-1           |
    - name: secret-2           |
tracking:                      |tracking
 enabled: true                 |  enabled: bool
 replicaCount: 1               |  replicaCount: int
 host: "0.0.0.0"               |  host: IP
 containerSecurityContext:     |  containerSecurityContext:
   runAsNonRoot: true          |    runAsNonRoot: true
# postgresql.arch              |
# standalone‘ or ‘repl         |
postgreSQL:                    |postgreSQL:
 arch: standalone              |  arch: standalone, repl
```

Figure 3.3. Schema generation from the *default Values* file used in the MLflow [63].

`bool`, `string`, `int`, `IP`, `[list]`, and `{dict}`, using regex-based substitution. (2) Replace enumerative fields replaced with lists including all valid options, extracted from annotations in the values file. (3) Lock predefined safe constants to fields critical to security, according to best practices for K8s resource specifications. For example, `securityContext.runAsNonRoot` can be locked to *true* [173]. Similarly, fields like *registry* and *image name* can be restricted to trusted values to mitigate risks like typosquatting attacks [185]. Thus, security-sensitive fields are locked with safe constants rather than placeholders, and any missing critical field is explicitly added. Figure 3.3 demonstrates an example of this process applied to the MLflow Operator.

Exploration of the configuration space. The values schema produced in the previous phase provides a generalized representation of possible configurations. However, it is still not ready for rendering with the Helm template (i.e., processing conditionals, loops, and placeholders in the template). The rendering process requires that only one value is indicated from the configuration space of enumerative fields in the schema. To address this, *KubeFence* performs the rendering multiple times, by exploring

different combinations of values in enumerative fields. Each combination leads to a variant of the schema (*values variant*).

At each iteration, *KubeFence* replaces enumerative placeholders in the schema with one of their valid values, while preserving placeholders for non-enumerative fields and constant fields. To avoid combinatorial explosion, *KubeFence* only explores a subset of configurations, such that each valid value for an enumerative field is covered in at least one generated variant. This process guarantees that the union of all variants covers all potential valid values from API requests, which should be allowed in the system by *KubeFence*. More specifically, at each iteration i, the process generates a new values variant by replacing each enumerative field with its i-th value. If an enumerative list has fewer options than the current iteration index, its last value is reused. The process iterates up to the length of the longest enumerative list. In the example of the schema in Figure 3.3 (right), this process generates two values variants, one for each option in the `arch` enumerative field.

Rendering of manifests. Once the *values variants* are generated by the previous phase, they need to be translated into Kubernetes manifests. These manifests are concrete representations of resource specifications, by resolving conditionals and loops and using actual values. These manifests will be the basis for generating the final security policies.

Each values variant is combined with the Helm template, using the `helm template` command. This command processes the template and values file to render a manifest. By the end of this phase, *KubeFence* produces a set of Kubernetes manifests, capturing all permissible configurations for the resources required by the K8s operator.

Generation of validators. The final step is to consolidate the generated manifests into a single *validator*, a YAML schema that defines all allowable configurations for K8s resources. This validator supports the enforcement of fine-grained security policies, by serving as a reference for validating incoming API requests.

Manifests are grouped based on the resource type (*kind*) (e.g., *Pod*, *Service*, *Deployment*) to ensure the resulting validator is organized and easily navigable. Each group represents the allowed configurations for a

specific Kubernetes resource type. Special placeholders (e.g., `string`, `int`, `hostIP`) from the manifests are retained to represent data types, enabling flexibility in configuration validation. Enumerative fields from multiple manifests are consolidated into arrays containing all valid values. Duplicate values are eliminated, while conflicting values are resolved by including all possible options in the array. Fields with constant, security-critical values (e.g., `securityContext.runAsNonRoot: true`) are directly carried over from the manifests without modification. This ensures that security best practices are enforced consistently across all configurations. Figure 3.4 shows a validator generated merging two manifests.

> The proposed approach automates the generation of fine-grained security policies for specific K8s workloads, by generating a YAML policy validator that captures all their allowable configurations, based on the systematic analysis of their Helm charts.

```
1  # Manifest sample 1                |
2  containers:                        |
3  - name: nginx                      |# Generated Validator
4    image: nginx:latest              |containers:
5    imagePullPolicy: IfNotPresent    |- name: string
6    ports:                           |  image: string
7    - name: string                   |  imagePullPolicy:
8      container:     IP              |  - IfNotPresent
9  # Manifest sample2                 |  - Always
10 containers:                        |  ports:
11 - name: nginx                      |  - name: string
12   image: nginx:latest              |    container: IP
13   imagePullPolicy: Always          |
```

Figure 3.4. Policy Validator generated from two manifests [63].

3.5.2 Enforcement of Security Policies

To enforce security policies and apply the generated validators, *KubeFence* employs Mitmproxy [76]. Mitmproxy is chosen for its capabilities in intercepting, inspecting, and modifying HTTP and HTTPS traffic, with support for SSL/TLS certificates. Mitmproxy is deployed as a Pod on each K8s control-plane node, positioned between the K8s API server and

clients (e.g., *kubectl, CI/CD pipelines, or Operators*). This ensures that all incoming API requests are intercepted and validated against the policy validator before reaching the API server.

In order to maintain a secure enforcement mechanism, the requests to the API server must not bypass the proxy (according to the *Complete Mediation* design principle [249, 262]). To this aim, the API server is restricted to accepting only certificate-based trusted connections, allowing only the proxy with a valid certificate to connect. Furthermore, since client-to-API server connections are encrypted, clients must trust the CA certificate of the proxy to enable traffic interception and decryption for inspection. Thus, proper certificate management is crucial for secure operations.

The core validation mechanism is implemented as a Python-based Mitmproxy plugin, which loads the YAML policy validator. When Mitmproxy intercepts an HTTPS request, the plugin parses the request body to extract the Kubernetes object, including the resource type (*kind*) and its specification fields, for validation. This validation process ensures that only authorized and correctly configured API requests are forwarded to the API server. The validation process operates as a hierarchical comparison, akin to a tree overlap, between the incoming manifest and the policy validator, interating across the requested K8s resources. The plugin extracts the *kind* field from the request to identify the resource type and verifies its presence in the validator. Then, it ensures that only fields explicitly defined in the validator are present in the manifest and validates that each field's data type matches the expected type specified in the validator. Enumerative fields and security-sensitive fields are validated against their strict list of allowed values in the validator. If the request complies with the validator rules, it is forwarded to the K8s API server unchanged. Otherwise, the plugin blocks the request, returning an HTTP error response to the client. Violations are logged with details of the offending field and the reason for denial, enabling auditing and forensic analysis.

3.6 Evaluation

This section evaluates *KubeFence* across three dimensions. First, we quantify the attack surface exposed by the Kubernetes API and demon-

strate how *KubeFence* can reduce unnecessary exposure by restricting access to unused endpoints and fields. Second, using a catalog of misconfiguration-based attacks and CVE exploits targeting specific API fields, we assess the effectiveness of *KubeFence* in mitigating these threats compared to native Kubernetes RBAC. Finally, we analyze the runtime overhead introduced by *KubeFence* for API request validation. A discussion of the limitations and potential biases of this empirical assessment, including the lack of a user study, is provided in Section 3.7.

3.6.1 Experimental Setup

We set up a K8s test environment replicating real-world deployment scenarios. The testbed includes a cluster using Kubernetes version 1.28.6, configured with a control-plane node and a worker node deployed on two distinct Ubuntu Linux 22.04.4 virtual machines. Both nodes are allocated 8 vCPUs and 16 GB of RAM, hosted on a machine with an *Intel Xeon E5-1620* 3.70 GHz CPU. *KubeFence* is deployed on the control-plane node, besides the API Server, using a container with Mitmproxy version 10.2.2.

This experimental analysis focuses on K8s Operators as a use case to demonstrate the feasibility and effectiveness of fine-grained workload-aware API enforcement. Since Operators are highly configurable, have extensive interactions with the K8s API [140], and are typically deployed using Helm charts, they are a compatible target for validating *KubeFence*. We select five Operators available on the Artifact Hub catalog [73], representing diverse categories of workloads commonly deployed in K8s clusters, including databases (*PostgreSQL* [37]), networking services (*Nginx* [36]), AI/ML applications (*MLflow* [35]), data streaming (*RabbitMQ* [38]), and security (*SonarQube* [39]).

3.6.2 K8s Attack Surface: Quantification and Reduction

The K8s API serves as the primary interface to a cluster, which the endpoints for users to query and modify resources. As a result, it represents a critical part of the attack surface for a cluster. We hypothesize that many workloads utilize only a small subset of this API, leaving a significant portion unnecessarily exposed and susceptible to exploitation. Reducing the accessibility of unused or unnecessary API endpoints and fields is a key

objective of *KubeFence*. In this experiment, we quantify the attack surface exposed by the Kubernetes API and evaluate how effectively *KubeFence* can reduce it by limiting access to non-essential endpoints and fields.

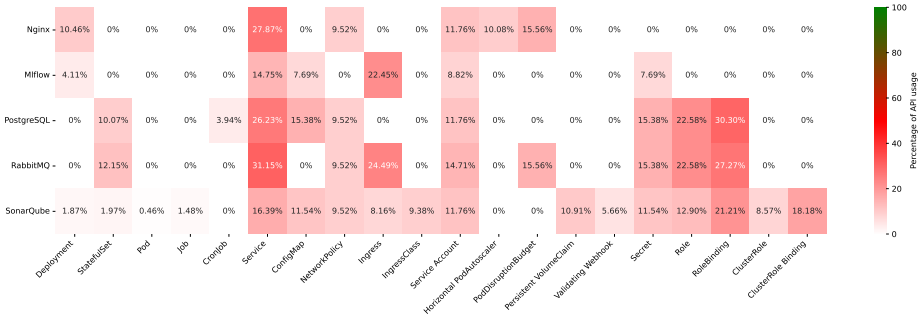


Figure 3.5. Percentage of API usage across workloads and endpoints [63].

To quantify the attack surface, we conducted a static analysis of the K8s API. This process involved counting the total number of endpoints exposed by the K8s API Server, and cataloging the configurable fields available for each resource type. Next, we analyzed the selected Kubernetes Operators to understand how real workloads interact with the API endpoints. By examining the validators generated through *KubeFence*, we identified the space of endpoints and fields that can potentially be used by each workload. This analysis provided a detailed understanding of K8s API utilization across different workloads and highlighted workload-specific behavior. The results are summarized in Figure 3.5, showing the percentage of fields utilized by each workload for each endpoint, relative to the total available fields.

Our findings revealed significant under-utilization of the K8s API by Operators, with numerous fields and endpoints remaining unused in practice. For instance, we observed that certain resources, such as *Pod* and *Job*, are entirely unused (i.e., 0% usage), by a substantial number of workloads. Other resources, such as *Service* and *ServiceAccount*, are actively used by all workloads, even if many of their fields are left unused. It is still worth blocking these unused fields since they contribute to the attack surface, potentially serving as entry points for exploitation, despite offering no functional value to the workloads. However, these resources cannot be

Table 3.1. Attack Surface Reduction Achievable by KubeFence vs RBAC [63].

Workload	Restrictable Fields		Attack Surface Reduction	
	RBAC	**KubeFence**	**RBAC**	**KubeFence**
Nginx	3747 / 4882	4751 / 4882	76.75 %	97.32 %
Mlflow	3883 / 4882	4826 / 4882	79.54 %	98.85 %
PostgreSQL	2906 / 4882	4711 / 4882	59.52 %	96.50 %
RabbitMQ	3676 / 4882	4708 / 4882	75.30 %	96.44 %
SonarQube	1012 / 4882	4772 / 4882	20.73 %	97.75 %

completely disabled, due to the frequent use of some of their fields. This leaves some residual risk of vulnerabilities in these APIs, as discussed in Section 3.7.

To evaluate the attack surface reduction achievable by *KubeFence* against RBAC, we analyzed the percentage of fields restrictable by each approach. RBAC restricts access to fields only when the entire resource type (API endpoint) is unused in the heatmap, meaning it lacks the granularity to filter individual fields within an allowed resource. In contrast, *KubeFence* can enforce restrictions on any unused field, even within partially-used endpoints. This makes *KubeFence* a strict superset of RBAC's enforcement, covering all fields RBAC could restrict while also providing additional reductions in the attack surface.

For the complete set of considered endpoints, we summed up the total configurable fields across all resources. For each workload, we computed the percentage of fields restrictable by RBAC and *KubeFence* as a measure of the attack surface reduction potentially achievable by the two techniques. Table 3.1 summarizes the results of this analysis. *KubeFence* consistently achieves a higher reduction in attack surface across all workloads, with improvements of 20.57%, 19.31%, 36.98%, 21.14%, and 77.02% across the five workloads, averaging 35% compared to RBAC. These results highlight that RBAC achieves lower attack surface reduction for workloads requiring multiple endpoints, as it cannot blacklist partially-used resources. In contrast, *KubeFence* can restrict unused fields within utilized endpoints, providing finer-grained protection.

Table 3.2. Catalog of K8s Malicious Specifications [63].

ID	Exploit/Misconfiguration	Targeted API Field	Ref.
E1	Activation of hostNetwork (*CVE-2020-15257*)	hostNetwork	[211]
E2	Abusing `LoadBalancer` or `ExternalIPs` (*CVE-2020-8554*)	externalIPs	[212]
E3	Command injection via `volume` and `volumeMounts` (*CVE-2023-3676*)	volumeMounts.subPath volumes.subPath	[216]
E4	Mount `subPath` on a file o `emptyDir` (*CVE-2017-1002101*)	volumeMounts.subPath	[209]
E5	Absent Resource Limit (*CVE-2019-11253*)	resources.limits	[210]
E6	Symlink exchange allow host filesystem access (*CVE-2021-25741*)	command	[214]
E7	Bypass of Seccomp Profile (*CVE-2023-2431*)	securityContext.seccompProfile.localhostProfile	[215]
E8	Privileged Containers (*CVE-2021-21334*)	securityContext.privileged	[213]
M1	Activation of hostIPC	hostIPC	[208]
M2	Activation of hostPID	hostPID	[208]
M3	Use Readonly Filesystem	securityContext.readOnlyRootFilesystem	[208]
M4	Running Containers as Root	containers.securityContext.runAsNonRoot containers.securityContext.runAsRootAllowed	[208]
M5	Allow Dangereous Capabilites to Containers	containers.securityContext.capabilities.add	[208]
M6	Escalated Privileges for Child Container Processes	securityContext.allowPrivilegeEscalation	[208]
M7	Custom SELinux `user` or `role`	securityContext.seLinuxOptions.user securityContext.seLinuxOptions.role	[208]

3.6.3 Catalog of Malicious Specifications

Malicious API requests pose critical security risks, by exploiting specific fields to achieve privilege escalation, unauthorized access to critical resources, and misconfigurations that may led to degradation of cluster availability or reliability.

To evaluate *KubeFence* against these threats, we built a catalog of 15 malicious specifications, comprising 7 misconfigurations and 8 malicious fields used by CVE exploits. These malicious specifications inject malicious values in Kubernetes manifests that can expose vulnerabilities or enable unsafe configurations, making them a practical subset for testing the effectiveness of *KubeFence* in mitigating attacks. Table 3.2 summarizes this catalog, identifying the targeted fields and providing references to their sources.

This catalog was developed by analyzing prior research [238], security blogs [271, 169, 29, 229], CVE disclosure [118], and Kubernetes security guidelines [173, 208]. Examples include enabling the `hostNetwork` field for host network sharing (CVE-2020-15257), exploiting `subPath` for host directory access (CVE-2017-1002101), and bypassing security profiles (CVE-2023-2431). We focus on CVEs from Section 3.2 that are exposed to malicious specifications from the K8s API interface, as these align with our threat model and the scope of API-level enforcement. We exclude CVEs that are we are unable to reproduce due to strict environmental prerequisites, which fall outside our experimental setup. For instance, Kubernetes clusters are affected by CVE-2023-5528 only if they use an in-tree storage plugin for Windows nodes.

3.6.4 Kubefence Effectiveness against RBAC

Misconfigurations and CVE exploits pose significant security risks in K8s (see Section 2.1.1). The native Kubernetes RBAC mechanism provides access control at the resource- and verb-level, but lacks the granularity to restrict individual fields within the resource specification. This experiment measures the effectiveness of *KubeFence* compared to RBAC, by evaluating its ability to *mitigate CVEs and misconfigurations*. To quantify this, we generate workload-specific policies for the five selected operators (listed in Section 3.6.1), and test whether *KubeFence* can block API-based misconfigurations and CVE exploits.

To test the enforcement mechanisms, we generated malicious API requests using our catalog of malicious specifications (Table 3.2). We inject the malicious fields in the catalog in resource types that support that malicious fields. For instance, the `spec.externalIPs` field is specific to *Service* resources. Other fields apply to relevant K8s resources, such as to *Pod* and higher-level abstractions like *Deployment*, *ReplicaSet*, *StatefulSet*, and *DaemonSet*.

Legitimate resource configurations were retrieved from Operator manifests, and malicious fields were injected into this configuration to create 15 distinct malicious manifests for each operator. For example, Figure 3.6 shows how the misconfigured `runAsNonRoot` field is injected into a *Deployment* resource from the Nginx Operator.

These malicious manifests were then submitted to the K8s API while

```yaml
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
      - name: nginx
        image: testImage
        securityContext:
          runAsNonRoot: false
```

Figure 3.6. Example of a malicious YAML manifest [63].

the respective workload-specific RBAC or *KubeFence* policy was in place. This process simulates realistic attack scenarios where a malicious client attempts to exploit CVEs or misconfigurations through the K8s API interface. The effectiveness of each enforcement mechanism was measured by recording whether each CVE exploit or misconfiguration attempt was mitigated.

Native K8s RBAC setup. We evaluated RBAC by configuring the K8s cluster with audit logging enabled, to capture API requests during the execution of an attack-free workload. Audit logs keep track of accesses to *API endpoint*, the *resource type* (e.g., Pods, Services), *verb* (e.g., get, create, update, delete), and the resource specification in API requests. Then, the audit logs were processed with the `audit2rbac` tool [183], which infers the minimum permissions required for a workload. This process generated five distinct YAML files, representing a tailored RBAC policy for each operator, based on the observed API interactions. Malicious manifests were applied to the cluster with RBAC policies in place. For each attack, we recorded the success or failure of the API request.

KubeFence setup. Then, we evaluated *KubeFence* by generating fine-grained security policies tailored for each workload, by analyzing the configurations required by the workloads. These policies were enforced using our proxy placed between the clients and the K8s API Server.

The same malicious manifests used for testing RBAC were applied against the *KubeFence* proxy. Each interaction of the Operators with the

API server was intercepted by the proxy, which validated the API request against the workload-specific policy (i.e., the validator). Moreover, our logs report the denied actions, and the malicious fields that triggered by filtering.

Experimental Results. Table 3.3 reports the mitigated CVEs and misconfigurations by RBAC and *KubeFence*, respectively. While RBAC did not block any of the attacks, *KubeFence* successfully blocked all of them.

Table 3.3. Mitigated CVEs and Misconfigurations by RBAC and KubeFence.

Workload	CVEs		Misconfigurations	
	RBAC	**KubeFence**	**RBAC**	**KubeFence**
PostgreSQL	0	8	0	7
Nginx	0	8	0	7
Mlflow	0	8	0	7
RabbitMQ	0	8	0	7
SonarQube	0	8	0	7

The results highlight that RBAC policies, even when tailored to workloads using tools like *audit2rbac*, lack the granularity to enforce restrictions on individual fields within resource specifications. Figure 3.7 illustrates an audit entry recorded during the deployment of the MLflow Operator, logging the creation of a *Deployment* resource. The generated RBAC policy effectively defined access at the resource level, specifying resource *kind*, *namespace*, *API group*, and allowed *verbs*. However, it omitted critical parameter-level details, such as `spec` fields "available" in the audit logs. This omission is not a limitation of *audit2rbac*, but rather an inherent limitation of RBAC policies, which do not allow specification at this level of detail. As a result, RBAC failed to prevent attacks that exploit features not needed by the operators, such as enabling *hostNetwork* or disabling *runAsNonRoot*.

By contrast, *KubeFence* successfully enforced fine-grained controls, blocking all attacks to misconfigurations and CVEs. For instance, it denied requests abusing the `subPath` field, as this parameter was not part of the configuration space defined in the Helm charts of the Operators. Fur-

Figure 3.7. RBAC policy generated (on the right) from an audited *create deployment* operation (on the left) [63].

thermore, legitimate workload actions were unaffected, demonstrating the precision and reliability of *KubeFence* in blocking unauthorized API request parameters without disrupting normal operations.

3.6.5 KubeFence Overhead

This section evaluates the runtime overhead introduced by *KubeFence* compared to the native K8s RBAC. The focus is on the online phase, where API requests are inspected and forwarded to the API server, as this directly impact cluster operations. The offline phase of *KubeFence*, which involves learning security policies, is excluded from this analysis as it does not affect runtime performance.

We measured the round-trip-time (RTT) latency for processing the full set of API requests generated during the deployment of the five selected operators. These requests are issued by the `kubectl apply` command from the client, and include all interactions with the API server to configure the resources defined by the Operator. The RTT latency was defined as the total elapsed time from issuing the *apply* command until the client received a response, indicating the API server had finished processing the requests. This experiment was conducted under two scenarios: first with native RBAC, and then with *KubeFence*. All requests were benign, as this experiment focused on normal operations rather than attack scenarios. To ensure statistical significance, we repeated the process 10 times per workload and computed the average latency and standard deviation for both RBAC and *KubeFence*. In addition, using the same workload, we evaluate the impact of *KubeFence* on system resources. To this end, we measured the CPU and memory usage of the proxy container, reporting the average and standard deviation over 10 repetitions.

Table 3.4 presents the average latencies and standard deviations for each workload, as well as the increase in latency introduced by *KubeFence* over native RBAC.

Table 3.4. RBAC vs KubeFence Average Request Latency

Operators	**RBAC RTT (ms)**	***KubeFence* RTT (ms)**	**Increase (ms, %)**
MLflow	211.0 ± 39.2	237.6 ± 37.5	+26.6 (12.61%)
Nginx	168.4 ± 25.7	210.4 ± 26.7	+42.0 (24.94%)
PostgreSQL	178.1 ± 16.1	213.6 ± 13.0	+35.5 (19.93%)
RabbitMQ	242.9 ± 16.6	307.6 ± 23.4	+64.7 (26.64%)
SonarQube	385.9 ± 14.0	470.5 ± 35.0	+84.6 (21.92%)

The results show that the additional latency introduced by *KubeFence* remains minimal, with absolute increases ranging from 0.0266 s to 0.0846 s. Even in the worst case, where the overhead reaches 26%, the total RTT latency remains well below 0.5 seconds, which is negligible for cluster management tasks such as workload deployment [46, 164]. Furthermore, this overhead impacts only operations initiated by external actors interacting with the K8s control plane, such as managing deployments and querying resource modifications, while internal API interactions by K8s components remain unaffected by *KubeFence*. Moreover, the application themselves (e.g., web resources served by Nginx) are unaffected. This minimal overhead is a worthwhile trade-off considering the enhanced attack mitigation capabilities and the attack surface reduction provided by *KubeFence*.

In addition, the impact of *KubeFence* on system resources was minimal. CPU usage increased only by 1.21% ($\pm$0.04), and memory consumption increased by 85.54 MiB ($\pm$0.25), which is a negligible overhead considering the security benefits.

The experimental results demonstrate that *KubeFence* effectively enhances Kubernetes security. It achieves significant attack surface reduction by restricting unused API fields and endpoints, addressing gaps in native RBAC that cannot enforce such fine-grained control. Moreover, *KubeFence* successfully blocks all tested misconfigurations and CVE exploits by precisely validating API requests against workload-specific policies, ensuring robust protection against real-world threats. Despite introducing a small latency overhead during API request validation, the impact remains reasonable for operations initiated by external actors and does not affect internal K8s operations. These findings establish *KubeFence* as a practical and efficient solution for securing K8s environments.

3.7 Limitations

Extensibility beyond Helm. The current implementation of *KubeFence* focuses on Helm-based workloads, using Helm templates to generate API security policies. While effective, this limitations its applicability to Helm deployments. However, the methodology of analyzing manifests to derive workload-specific security policies can be easily extended to other deployment mechanisms, such as Kustomize or raw YAML manifests. By adapting the parsing and policy generation processes, *KubeFence* can ensure consistent security enforcement across diverse deployment workflows.

Scope of attack surface hardening. The primary objective of *KubeFence* is to *reduce the Kubernetes attack surface*, denying unnecessary and risky features on a per-workload basis. Despite this, it does not claim to eliminate CVE exploitability or misconfiguration risks entirely. The solution leverages the client configuration space to infer which API endpoints and fields allow, and best practices guidelines to infer which critical field to lock to safe values. This significantly mitigates opportunities for attackers to exploit malicious configurations irrelevant to the workload.

KubeFence is based on the idea of blocking code not used by common workloads, which can be difficult to apply for some users. It is possible that uncommon, but legitimate workloads are blocked by such restrictive security policy. In general, false positives are a challenge for any filter-

ing approach, as the same issue is faced by admins that manage firewalls, IDS, and similar tools. *KubeFence* mitigates false positives by tailoring policies to K8s operators, which are becoming a popular approach to manage clusters. In such use cases, features can be restricted with very high accuracy. However, in other use cases, it may be difficult to anticipate which features should be allowed. Still, some enterprises may still want to block uncommon workloads, and enable them only after more careful scrutiny.

It is also possible that *KubeFence* does not restrict interfaces that are prone to vulnerabilities, in the case that these interfaces are used by legitimate workloads, in order not to disrupt them. These interfaces represent a residual risk, which has to be handled through other complementary strategies. One approach is to adopt anomaly detection methods on API calls, which can identify misuses and exploitation attempts of the features [174]. Another strategy is to perform more thorough testing, such as fuzzing [91], to identify vulnerabilities in the residual attack surface.

KubeFence does not validate the functional correctness of Helm charts when they introduce unnecessary features or omit required ones. Instead, it enforces the stated resource definitions as provided. Ensuring correctness in such cases is an orthogonal problem and fall outside the scope of *KubeFence*. External YAML validation tools (e.g., KubeLinter [270], Checkov [237]) can be used before policy generation to address this problem.

Furthermore, *KubeFence* does not address risks arising from compromises in the Kubernetes Operator catalog through supply chain attacks, where malicious Operators could inject unsafe configurations. In such cases, the responsibility for addressing these risks lies with workload developers.

Evaluation Bias and User Studies. The empirical evaluation of *KubeFence* was conducted by the same researcher who designed and implemented the system, and who was aware of the misconfigurations and CVE-based attacks used in our catalog. This setup enabled us to explore the limits of the approach in a controlled way, but it may also introduce a bias in how workloads and malicious specifications are selected, configured, and interpreted. In particular, the current evaluation does not demon-

strate whether practitioners who did not develop *KubeFence* would be able to (i) specify appropriate policies, or (ii) run representative workloads to drive automatic policy generation, in settings where vulnerabilities are not known in advance. A systematic user study with engineers and cluster administrators, for example measuring effort, correctness, and false positive rates when using *KubeFence*, is therefore an important direction for future work beyond this thesis.

Performance Optimizations. While the overhead introduced by *KubeFence* is negligible for most cluster management tasks, with latency increases ranging from 0.0266 to 0.0846 s, it could still impact performance-critical of real-time deployment scenarios. The current implementation relies on a proxy Pod to intercept, validate, and forward external API requests to the API server, which adds network latencies. To address this, *KubeFence* could be integrated directly into the API server, eliminating the forwarding delays and leaving only the validation cost. Although it requires modifications to the API server codebase, this approach offers a viable path to optimize enforcement efficiency for more demanding use cases. From a deployment perspective, both the current proxy-based design and a tighter integration into the API server are compatible with how Kubernetes distributions already ship additional security components (e.g., admission controllers and policy engines). This suggests that integrating *KubeFence* as part of a Kubernetes distribution, either as an optional add-on or as a built-in hardening module, is technically feasible, and we leave a full exploration of this integration path to future work with distribution maintainers.

3.8 Discussion

The extensive and feature-rich Kubernetes API, coupled with coarse-grained authorization mechanisms such as RBAC, exposes clusters to misconfigurations and exploitable features that significantly expand the orchestration layer attack surface. This chapter addressed this problem through *KubeFence*, a runtime enforcement system that automatically generates and enforces fine-grained API security policies tailored to individual workloads. By restricting workloads to their minimal required API inter-

actions, Kubefence enforces workload-specific least-privilege policies and demonstrates that systematic attack-surface reduction is achievable even in dynamic, declarative environments like Kubernetes. Beyond its specific application, Kubefence embodies a broader principle central to this dissertation: effective attack-surface reduction requires enforcement mechanisms that operate precisely where software complexity manifests. In orchestration systems, that critical interface is the management API; at the operating system level as runtime enforcement of syscall capabilities; in the software supply chain as dependency-level behavioral constraints. The next chapter builds on this same philosophy, shifting from the orchestration API to the inter-process communication (IPC) surface, another critical interface where complexity and privilege intersect. There, we explore how full-system emulation and gray-box fuzzing can uncover and mitigate vulnerabilities in industrial, binary-only environments, extending the scope of attack-surface reduction from orchestration to communication boundaries.

Chapter 4

Fuzzing of IPC Surface

4.1 Overview

Across modern software systems, from cloud-native platforms to embedded and industrial deployments, security-critical functionality increasingly relies on complex, stateful software components that are difficult to test exhaustively. As discussed in the orchestration domain, large-scale automation and rich interfaces amplify the impact of latent vulnerabilities; similar effects arise in industrial and embedded systems, where failures at low levels can propagate across tightly coupled components. In these contexts, zero-day vulnerabilities continue to undermine system reliability. Fuzz testing is one of the most effective strategies to uncover such vulnerabilities [302, 191, 99], yet existing fuzzers commonly assume that the target can be instrumented and rebuilt to include a fuzz driver. This assumption breaks down in industrial settings, where proprietary or legacy toolchains often prevent source instrumentation or binary rewriting. Consequently, many embedded targets remain untested or are evaluated only through limited black-box methods.

This chapter presents *FuzzBox*[1], a novel fuzzing framework that integrates fuzz testing with full-system emulation to overcome these constraints. Built atop QEMU, *FuzzBox* transparently intercepts target function calls, injects mutated inputs, and monitors execution feedback, without requiring source access, recompilation, or hardware-specific tracing support.

[1] Available as open-source at `https://github.com/dessertlab/FuzzBox`

This design achieves cross-architecture portability and extends fuzzing to domains traditionally excluded from coverage-guided testing, such as proprietary real-time OSes and legacy firmware. We evaluate FuzzBox on a proprietary embedded hypervisor (Wind River VxWorks MILS [204]) and Linux-based IoT firmware. Compared to black-box fuzzing, FuzzBox achieves a 2× increase in throughput and a 58% improvement in coverage, uncovering faults that previous methods could not reach.

4.2 FuzzBox Design

The design of *FuzzBox* is based on the following design principles, in order to address the challenges discussed in the previous section:

1. **Non-Intrusive**: The tool should avoid any modification of the target software, such as the introduction of a fuzz driver to trigger the target.

2. **Toolchain independency**: The tool should avoid depending on the availability of build toolchains able to perform code instrumentation. It should still be able to collect information about code coverage and should support binary-only (e.g., closed-source) targets.

3. **Hardware independency**: The tool should avoid any reliance on hardware features, in order to support a broad applicability across different embedded systems.

Figure 4.1 shows an overview of the workflow of *FuzzBox*. The architecture is flexible and configurable, supporting various operations for fuzzing. Users initiate the process by specifying a *configuration* for fuzzing, such as the entry points of the target to be fuzzed (step ① in Figure 4.1). Following this, *FuzzBox* employs the QEMU emulator [48] to execute the unmodified binary of the target software within a guest VM in full system mode. Leveraging QEMU's introspection capabilities, *FuzzBox* intercepts and fuzzes configured target functions by profiling and manipulating the guest VM state (steps ② , ③ , ④ in Figure 4.1). While executing fuzzed operations, *FuzzBox* actively monitors the guest for abnormal behavior. Furthermore, for effective input generation, *FuzzBox* transparently gathers coverage information within the QEMU emulation engine (steps ⑤ ,

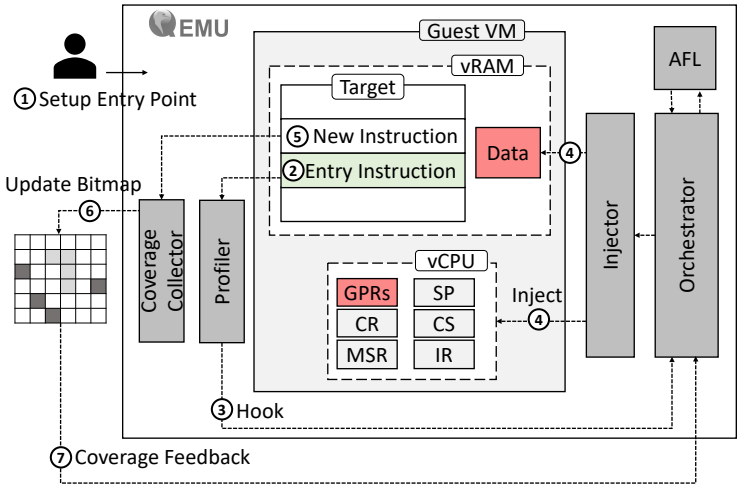


Figure 4.1. An Overview of the Proposed Approach *FuzzBox* [62].

⑥ in Figure 4.1). This design blends fuzzing into the emulation environment, which allows us to address challenges in fuzzing industrial systems, and enables the applicability of *FuzzBox* to a broad range of embedded systems based on different CPU architectures and hardware platforms.

It is worth noting that *FuzzBox* enables fuzzing binary-only industrial firmware, including transparent instrumentation, and crash detection. It does not propose new strategies for generating fuzz inputs (e.g., input-mutation and scheduling heuristics), as these aspects are orthogonal to our proposed solution.

4.2.1 Interception

The *FuzzBox* fuzzing process is designed to intercept the invocation of a configured *target function* during the normal execution of the target software and inject a new fuzz input. In this context, a *fuzz input* refers to the specific values injected as input parameters of the target function, affecting the execution path of the target and potentially discovering vulnerabilities. The interception capability is carried out by two components in *FuzzBox*, the *Profiler* and the *Injector*.

Profiler. It observes the execution of the target according to configured events. An *event* of interest is the invocation of a target function, which represents a fuzzing attack surface (e.g., an I/O function handling external input). Such functions serve as the entry point for *FuzzBox*'s fuzzing workflow. During interception, the profiler also collects input parameters, enabling seed collection. It can hook the execution either before invocation (*pre-invocation*) or after (*post-invocation*), depending on the fuzzing strategy. Event configuration is further discussed in Sec. 4.2.4.

Injector. It manipulates the state of the guest VM to fuzz function parameters. Fuzz inputs can be injected into registers or memory, depending on the calling conventions of the target architecture. This design allows *FuzzBox* to fuzz both input and output parameters across a wide range of function types. For instance, input parameters of "*send*" functions must be fuzzed before invocation, while output parameters of "*receive*" functions are fuzzed after invocation, to avoid being overwritten during execution. Once fuzz input is injected, the VM resumes execution. By intercepting function calls and injecting fuzz input transparently, *FuzzBox* does not require an external fuzz driver or modifications to the target, thus maintaining a completely non-intrusive fuzzing approach consistent with the first design principle.

4.2.2 Feedback

Coverage Collector. It supports *FuzzBox*'s fuzz input generation by transparently gathering *code coverage* from the target during execution. This feedback is crucial for effective coverage-guided fuzzing, prioritizing fuzz inputs that explore new code paths while discarding those that do not. The collector analyzes two common types of coverage, *Basic Block Coverage* and *Edge Coverage*, as typically used by gray-box fuzzers. Basic blocks (BBs) represent sequences of instructions bounded by control flow transfers (e.g., jumps, calls, or returns), while edges reveal block-to-block transitions, providing insight into taken paths. Unlike traditional fuzzers that instrument code at compile-time, *FuzzBox* maintains unmodified source and binaries. Coverage is instead observed at the system level, including not only application code but also other components (e.g., the kernel) within the target. These components may introduce

non-deterministic control flow (e.g., from context switches or timer interrupts), creating spurious coverage. To mitigate this, *FuzzBox* conducts a pre-analysis phase with repeated unmodified inputs, blacklisting blocks and edges triggered spuriously. Since fuzzing may involve only part of a binary, the bounds to trace are configurable.

Crash Detection. *FuzzBox* enables tracing the occurrence of failure events as additional feedback data, enhancing the input generation of AFL. This feature leverages the same Profiler described in Section 4.2.1, which can be optionally configured to intercept crash or error handling functions (e.g., *exit*, *abort*, *assert*). For instance, in MILS-based environments, specific functions responsible for managing virtual board suspension during crashes can be traced.

This novel approach aligns seamlessly with the second and third design principles, emphasizing the flexibility and independence of *FuzzBox* from specific build configurations and diverse embedded hardware setups.

4.2.3 Fuzzing Orchestration

The orchestrator drives the fuzzing logic in *FuzzBox*. It serves as a command and control component designed to coordinate the profiler, injector, and coverage collector components, alongside the fuzzer engine (i.e. AFL). *FuzzBox* implements two different modes of operation: *seed recording* and *intercept-and-fuzz*.

Seed Recording. This mode supports the collection of *fuzz seeds*, that is, initial inputs to be mutated by the fuzzing process. The *orchestrator* uses the *profiler component* to intercept invocations of a configured target function during a regular execution of the target software. The frequency and duration of recording can be configured by the user. The orchestrator gathers a fuzz seed for every interception, which consists of the parameters passed to the intercepted function, and saves them in a queue. Additionally, for each fuzz seed generated, the tool supports the saving of an *initial fuzz state* for snapshot-based fuzzing, which is represented by the VM state at the time of intercepting the function call.

Intercept-and-Fuzz. This operation mode is designed for event- and time-triggered industrial applications (e.g., MILS-based applications), where software operates based on cyclic internal tasks, which periodically invoke the target function [145]. In this mode, the fuzzing workflow intercepts at run-time the periodic invocations of the target function, and replaces the original inputs with fuzz inputs. The *orchestrator* selects an input to be fuzzed from the pre-recorded *fuzz seed*. The *fuzzer engine* applies mutations to the seed (e.g., bit flips, byte flips, arithmetic operations, and havoc as in the AFL fuzzer), generating a set of *fuzz inputs*, which are enqueued. Because these mutations are applied to seeds collected from real executions, many fuzz inputs remain syntactically well-formed with respect to the underlying protocol or data structure, while others intentionally violate constraints or stress boundary conditions. As a result, *FuzzBox* exercises both robustness to malformed data and nuanced behaviors under rare but still legal inputs, rather than being limited to purely invalid test cases. This enables the detection of a range of faults, including robustness issues (e.g., memory safety violations, crashes, hangs) triggered by malformed inputs, as well as deeper logic errors that arise only under seldom-occurring but valid combinations of parameters, making the approach effective both for stress testing and for uncovering subtle semantic defects. As the target software runs on the guest VM, the *profiler* intercepts regular invocations of the target function. For each invocation intercepted during the fuzzing campaign, the orchestrator selects a fuzz input from the queue, and injects it using the *injector*. Throughout the campaign, the *orchestrator* utilizes the *coverage collector* to inform the fuzzer engine about new execution paths. Fuzz inputs exploring new paths are marked as interesting and included in the queue for mutation. Additionally, during the campaign, the orchestrator also uses the profiler to identify anomalous behaviors, such as VM crashes or hangs by tracking abort, exit, or assert functions. The detected states serve as test outcomes for further analysis. The user terminates the fuzzing campaign.

4.2.4 Configuration

Since MILS systems can support many diverse and complex system configurations, *FuzzBox* is highly configurable to adapt and perform fuzzing on different target operations. Then, it requires specific configuration

parameters for effective usage. The target function for fuzzing can be kernel system calls, or library function calls in a user-space application. For instance, when fuzzing MILS-based applications (e.g., an application gateway), *FuzzBox* can focus on SIPC kernel primitives for inter-partition communication. Users need to identify the target function and configure *FuzzBox* with the associated symbol. We assume that information about the target function is available to the user through product documentation and specifications (e.g., documentation from the vendor of the MILS kernel about kernel primitives), and technical standards (e.g., the POSIX standard used by UNIX, Linux, and other OSes). *FuzzBox* then parses the target binary (e.g., in ELF format) to automatically identify the memory address where target symbols will be loaded, using the *GNU nm* utility. The locations of the symbols become the entry points for fuzzing. If symbol information is stripped from the binary, the user requires additional solutions to identify the addresses, which have been developed in other studies and are outside the scope of this work. The user can adopt reverse engineering tools like Fuzzable [2] to automate the discovery of the target function address through static analysis on binaries.

The user can specify which input parameters of the function call are going to be fuzzed. Fuzzing often targets complex input data (e.g., large data structures), which are typically passed to functions as pointer parameters. In our current design, the tool dereferences the indicated input parameter as a pointer, and corrupts the pointed memory area. The size of the memory area to be corrupted can be specified in two ways: it can be statically configured by the user (e.g., when fuzzing a `struct` parameter used for passing complex data structure in C), or it can be derived from another input parameter provided to the function (e.g., C functions often pass byte arrays along with an integer parameter indicating the array size), which the user can also configure.

FuzzBox is adaptable to various CPU architectures, by allowing the user to configure the calling conventions and register usage specific to the target architecture. For example, the PowerPC architecture uses the general-purpose registers GPR3-GPR10 to pass the first eight parameters, while any remaining parameters are passed on the stack [17]. Similarly, ARM and MIPS pass the first four parameters in registers R0-R3 and A0-A0, respectively. In addition, the tool can retrieve the return address of

a function call at the time of the invocation, such as, by reading the Link Register (LR) register in the ARM and PowerPC architectures, or from the Return Address Register (RA) in MIPS architecture. This architectural awareness ensures that *FuzzBox* accurately intercepts functions and parameters and injects the fuzz input.

In addition, during the configuration phase, utilizing the Code Coverage Collector component allows running the target system in idle conditions, facilitating the automatic gathering of basic blocks to blacklist for the coverage feedback, as discussed in Section 4.2.2.

The configuration of the information described in this section represents the only manual effort required to use *FuzzBox*. The calling conventions need to be configured just once for each architecture, and *FuzzBox* comes with pre-defined profiles for popular architectures including PowerPC, ARM, and MIPS. The configuration of the target function is performed only once for the target binary, and can potentially be reused across different applications that use the same kernel and functions (e.g., the SIPC primitives of a MILS kernel product). While setup time varies depending on the target system and user expertise, configuring a new target typically requires between 30 minutes and a few hours. In our experience, most of this time is spent reviewing target specifications and header files, and is significantly less than the effort needed to develop a dedicated fuzz driver to the target application, which is a key limitation towards the adoption of fuzzing [42, 146, 149].

4.3 Implementation

FuzzBox is built upon QEMU emulator version 7.0. We chose QEMU due to its broad support for several CPU architectures, including x86, ARM, MIPS, and PowerPC [18]. Additionally, it offers support for various physical boards and peripherals, and its inherent extensibility further adds to its potential. In fact, we have added customization gathered from Adacore [10] (a partner of VxWorks) to support the MPC8548E development board, which is the System-on-Chip on which VxWorks MILS executes. Our implementation of *FuzzBox* is integrated into the codebase of the emulation engine. The fuzzing engine within *FuzzBox* is developed using the state-of-the-art *libAFL* library [110], a modular and configurable

library that incorporates AFL algorithms. The details of each component are briefly described below.

Profiler. It is implemented as a TCG plugin [19], which subscribes to events generated by the QEMU translation process. QEMU includes Tiny Code Generator (TCG), which translates guest VM code into an intermediate representation and then into native code for the host. This translation process enables both emulation of different architectures and the insertion of tracing/debugging hooks. The profiler registers a callback for each translated basic block; if a block includes the instruction marked as the fuzzing entry point (from the configuration), a second callback hooks execution of that instruction. Upon entry-point execution, the plugin pauses the VM to profile the VM state and collect function parameters. Since TCG plugins cannot directly access guest state [19], we extended the QEMU API with two custom functions: `get_cpu_register()` and `vcpu_read_phys_mem()`, providing access to guest vCPU registers and memory. The profiler can store this data in host-side files to facilitate seed collection. To support both pre- and post-invocation interception, the profiler hooks either the first instruction of the function or the return address stored in the LR register at call time.

Injector. It is implemented as an additional configurable callback within the same TCG plugin, triggered sequentially after the profiler. To fuzz target function parameters, we expanded the QEMU API with two additional functions that override guest registers and memory. Depending on the calling convention of the architecture, fuzz inputs are placed in the appropriate registers, stack slots, or memory locations. For stack-based parameters, the injector dereferences the stack pointer; for pointer-based parameters, it dereferences the register or stack location holding the pointer. With this design, calling conventions must be configured only once per architecture, and then reused across applications. Injections can be performed pre-invocation or post-invocation depending on the function type, as described in the design.

Coverage Collector. It is implemented by integrating with QEMU's translation engine. Specifically, it builds on AFL's QEMU mode, a patch

originally designed to capture edge traces in user-space Linux binaries. *FuzzBox* extends this mode to operate in QEMU's full-system emulation, thereby enabling coverage collection for embedded applications, kernels, and firmware. The implementation uses the vCPU program counter and the mapping between basic blocks and translation blocks (TBs) within TCG to gather both block and edge coverage during execution.

Fuzzing Orchestrator. It runs as a separate thread in the QEMU process. This thread controls and synchronizes all other architectural components, as well as the fuzzing engine (which runs in a second QEMU thread). Additionally, it handles the parsing of user parameters from the command-line interface (CLI) and configures *FuzzBox* accordingly.

4.4 Evaluation on Industrial Embedded Systems

In this section, we evaluate *FuzzBox*'s effectiveness in enabling modern gray-box fuzzing for binary-only targets that rely on proprietary toolchains and platforms, as found in industrial systems.

Our evaluation is structured around two primary aspects: fuzzing depth and performance. Fuzzing depth is assessed through bug discovery and code coverage. Bug discovery measures the ability of *FuzzBox* to uncover potential vulnerabilities, while code coverage measures the extent to which the tool explores execution paths in the target, indicating the comprehensiveness of its analysis. Performance is evaluated in terms of throughput, defined as the number of fuzz inputs processed per second. Since fuzzing requires the generation and execution of a large volume of inputs, high throughput is essential to maintain efficiency and scalability.

4.4.1 Evaluation Targets

In our experiments, we consider the Wind River VxWorks MILS industrial platform. These experiments consider MILS-based applications as targets of evaluation. The MILS-based targets selected for our evaluation are inspired by a real-world defense domain gateway application, which involves client-server communication where *parsing libraries* are used to validate and filter exchanged payloads. These systems are of critical im-

portance in security-sensitive environments, but testing them poses significant challenges (see Section 2.2.2), and existing fuzzers cannot be applied effectively. To replicate the functionalities of a defense-domain gateway we used the VxWorks MILS hypervisor alongside open-source (vulnerable) software components, since we are unable to disclose information about the actual proprietary application. MILS guard applications perform input validation and parsing of data inputs, in order to inspect the data before they are forwarded to another network.

Therefore, we consider libraries that parse structured data formats as fuzzing targets. We selected three POSIX-compliant open-source parsing libraries, with minimal external dependencies, in order to facilitate the porting of these libraries to VxWorks MILS. We consider JsonParser [13], responsible for parsing serial data as JSON strings; SendMail [12], a preprocessor for email addresses aimed at normalization; TinyExpr [14], utilized for parsing and evaluating mathematical expressions. These targets perform parsing and validation of diverse structured data formats, similarly to real-world security guard applications [248]. Vulnerabilities in such libraries could compromise the security guarantees provided by the MILS architecture. We compiled binary executable using the original toolchain from the MILS product, without any change to the build process. We performed the experiments without assuming any access to the source code and to the build toolchain. The system was run on a QEMU virtual machine emulating the *WindRiver SBC8548E* development board, which uses a PowerPC MPC8548E CPU with 1 GB RAM.

Table 4.1 lists the selected targets, their type, and the CPU architectures that were emulated during the experiments. We conducted experiments on a host system with an *AMD Ryzen 5 3600 6-Core 3.98 GHz* processor and 12GB RAM.

Table 4.1. MILS-based Applications Targeted in the Evaluation.

Target	**Description**	**Architecture**
json [13]	Parses serial data as JSON strings	PowerPC
sendmail [12]	Pre-processes email addresses	PowerPC
tinyexpr [14]	Parses mathematical expressions	PowerPC

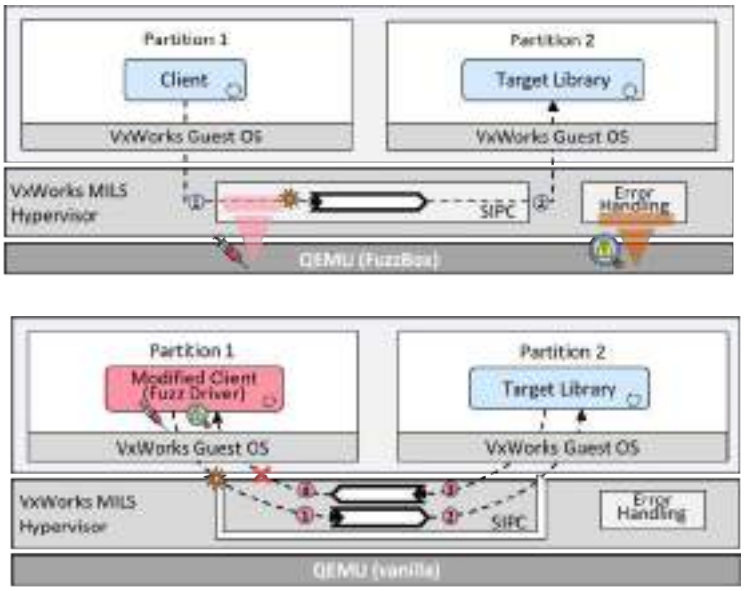


Figure 4.2. *FuzzBox* (top) and *baseline* (bottom) setup to test MILS systems [62].

4.4.2 Experimental Setup

To evaluate and compare *FuzzBox* coverage-driven approach against traditional approaches, we configured two distinct testing setup.

FuzzBox Testing Setup. We configured VxWorks MILS to implement the MILS gateway use case in a simplified setup. Figure 4.2 (top) shows the *FuzzBox* testing setup, involving two VBs running the VxWorks Guest OS. These VBs communicate through SIPC channels through the VxWorks MILS hypervisor. The client uses the `sendMessageSIPC()` kernel primitive to iteratively send messages to the second VB (step ①).

The server uses the `receiveMessageSIPC()` routine to receive messages and invokes the target library to process them (step ②). In addition, in this setup, the MILS hypervisor utilizes `schedSuspendVb()` primitive to manage the suspension of VBs during crashes, ensuring fail-safe behavior. We compiled all of the described components and workflows in a

binary package that we executed on top of a *FuzzBox*-enabled VM. In this testing configuration, *FuzzBox* operates in *intercept-and-fuzz* mode with edge coverage feedback enabled. Configuring *FuzzBox* for a specific target to fuzz is a straightforward process, ensuring a user-friendly experience with minimal manual effort. The user identifies the target function (`sendMessageSIPC()`), specifies the parameters to fuzz and their position (3rd and 4th parameters as message and length), selects the CPU architecture (PowerPC), and chooses the crash detection function (`schedSuspendVb()`).

Baseline Testing Setup. The Baseline setup adopts a traditional configuration, with a modified client that acts as a custom fuzz driver and failure detector through liveness checks. In this setup, we consider the popular libAFL library [110] as reference fuzzer. We emphasize that both the *FuzzBox* and the Baseline setups use the same LibAFL-based fuzzing engine, with identical input mutation logic and seeds. This ensures that any observed differences in fuzzing depth or throughput can be attributed to architectural differences, particularly the enhanced introspection and feedback loop enabled by *FuzzBox*, rather than to different fuzz input generation strategies.

We remark that we could not consider other state-of-the-art fuzzers for our evaluation on MILS-based applications, as they lack support for niche operating systems and architectures used in industrial systems (see the challenges identified in Section 2.2.2).

Therefore, we had to develop a custom fuzz driver, using the libAFL library to orchestrate fuzzing (e.g., generating mutations, enqueueing fuzz inputs, etc.). Figure 4.2 (right) shows the Baseline configuration. Again, two VBs communicate through an SIPC channel. In this setup, a black-box fuzz driver is integrated into the application, taking the role of the client. It sends fuzz inputs to the server in the second VB, which is the same as the previous setup (steps ① and ②). Since there is no direct way for the client VB to check the status of the server VB, the setup also includes an additional SIPC channel (not needed by the MILS-based gateway application) for response messages from the server to the client. The fuzz driver checks the response queue from the server, by setting a timeout to detect failures (steps ③ and ④). The resulting binary is executed on

a vanilla QEMU, with all *FuzzBox* components disabled except for the coverage collector, used for experimental evaluation purposes. *Baseline* is limited to collect coverage statistics, without using feedback to guide the fuzzing process. Without our tool, this setup is only feasible when it is possible to recompile the target application, with an additional effort to adapt the fuzz driver.

4.4.3 Fuzzing depth

FuzzBox enables coverage-guided fuzzing for binary-only industrial targets, where traditional coverage measurement is challenging due to limited introspection capabilities. To evaluate the fuzzing depth achieved by our proposed technique, we compared its code coverage and bug finding capabilities against traditional black-box fuzzing [160].

In both configurations, the *FuzzBox* architecture was used to collect code coverage metrics. However, in the black-box fuzzing setting, coverage data was gathered solely for measurement purposes and did not influence the fuzzing input generation. In contrast, *FuzzBox* setup used the coverage feedback to guide the input generation process.

To assess the bug finding effectiveness of *FuzzBox* against the *Baseline*, we built two buggy versions of each of the three POSIX-compliant libraries introduced in Section 4.4.1, utilizing two different injection strategies, resulting in six evaluation targets. The "*easy*" buggy versions of the libraries include a bug injected into frequently-exercised and easy-to-reach path of the application, discoverable with straightforward fuzz inputs and minimal mutations. Conversely, the "*hard*" buggy versions include a bug injected deeper within the application logic, requiring a more intricate fuzz mutations for discovery. This approach is inspired by the well-known LAVA approach [90], which injects bugs into carefully-selected paths of a program to challenge bug finding tools. We ensured that each vulnerability has an independent input trigger condition. The injected bugs cause a memory violation, which is representative of real-world memory safety vulnerabilities in embedded systems [287, 142, 116]. Specifically, the bug corrupts a register of the TSEC Ethernet controller in a way that triggers a read from a privileged memory region. This region lies outside the allowed address space of the application code running on the virtual board, and any access to it results in a crash of the VB itself. This fault reflects a

common class of vulnerabilities in embedded systems, where misconfigured or attacker-controlled peripheral accesses can violate memory protection boundaries enforced by the RTOS and hardware MMU. These bugs mirror known vulnerability patterns in MILS-based systems, such as CVE-2019-12255 and CVE-2019-12264, where malformed inputs to the network stack of VxWorks OS led to memory accesses outside the permissible range, involving control and status registers mapped to privileged memory.

Coverage Analysis. In the first experiment, we evaluated the capability of *FuzzBox* to achieve high code coverage compared to the *Baseline* approach. Since the two approaches may trigger a bug in the target within different time windows, with one potentially requiring less time than the other, we standardized the comparison by fixing the time window to the shorter duration required to trigger the bug between the two approaches. Then we measured edge and basic block coverages for both approaches in the selected time window. Table 4.2 shows the total number of edges and blocks discovered by the two techniques in the reference time windows, across the selected targets. Furthermore, both coverage growth curves over time are plotted in Figure 4.3.

Table 4.2. Edge (and BB) coverage achieved by *FuzzBox* and *Baseline* in a fixed time window.

Target	T.W. (s)	FuzzBox	Baseline	Impr %
json_easy	2197.53	2117 (837)	1913 (801)	+10.66%
json_hard	895.09	746 (353)	327 (159)	+128.13%
sendmail_easy	85.22	188 (51)	84 (39)	+123.80%
sendmail_hard	282.68	2520 (1455)	2559 (1434)	-1.52%
tinyexpr_easy	6.12	653 (426)	622 (413)	+4.98%
tinyexpr_hard	11913.52	2851 (782)	1600 (654)	+78.18%

Each test concludes when a crash is identified in the target or when coverage saturation is observed. Indeed, using coverage as feedback to generate fuzz inputs improves the discovery of new paths in the program. Examining the final edge coverage achieved by *FuzzBox* and the *Baseline*, our solution demonstrates changes of +10.66%, +128.13%, +123.80%, -1.52%, +4.98%, and +78.18% across the six targets, with an average improvement of 57.37%. Furthermore, analyzing the coverage curves over time, in most cases, *FuzzBox* achieves higher edge and basic block coverage more rapidly

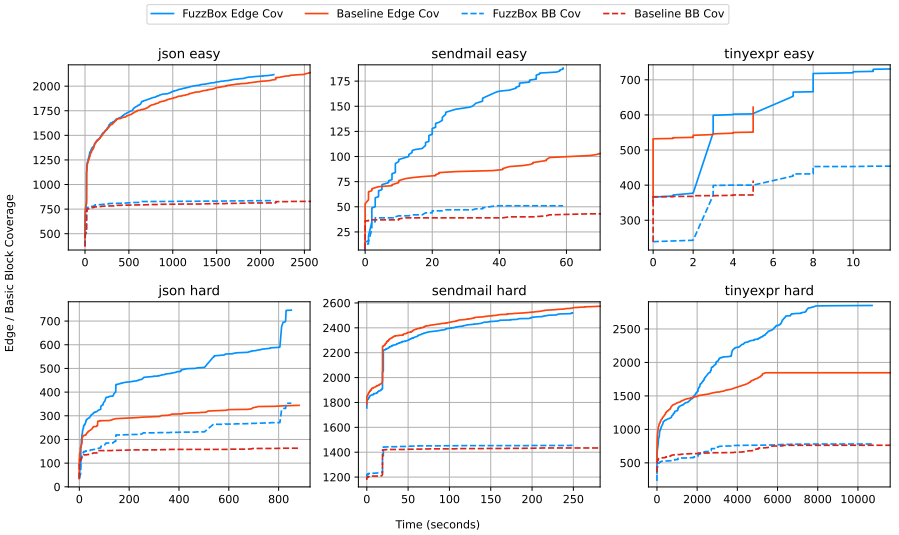


Figure 4.3. Coverage growth curves over time for MILS-based application targets [62].

than the *Baseline*.

Bug Analysis. In a second experiment, we evaluated the bug discovery capabilities of *FuzzBox*. Table 4.3 presents the results for the six targets. For each case, it reports the time to crash (TTC) in seconds and the number of iterations required (i.e., unique mutations) to cause a crash. Depending on the target, different initial seeds were used, as documented in the table. TThe determinism of AFL ensures the consistent generation of the same input sequences when using the same seed in multiple test repetitions, which makes the experiments reproducible for a given configuration. However, we note that our reported times-to-crash and coverage metrics are based on single executions per configuration; repeating the experiments with multiple runs and aggregating results statistically would further strengthen the empirical analysis (see Section 4.6). When applied to the "*hard*" targets, *FuzzBox* consistently outperforms the *baseline* approach. In two cases, the *baseline* fails to detect the bug within a 1-hour time window. This highlights the benefit of leveraging code coverage,

Table 4.3. Discovered bugs and TTC in *FuzzBox* and *Baseline* for MILS-based applications.

Target	Seed	Fuzz Input	TTC (s)	Iter.	Approach
json_easy	`AA`	`{}`	2197.53	95481	*FuzzBox*
		`{}`	4987.66	397629	*Baseline*
json_hard	`{bbbbbbb}`	`{"^?\bpp`	895.09	53100	*FuzzBox*
		`{"bi\b`	921.42	71173	*Baseline*
sendmail_easy	`ABCDEFG`	`(B((^],\Û^i(BÛ^i(B^c×œ`	85.22	2817	*FuzzBox*
		-	> 1h	200000+	*Baseline*
sendmail_hard	`AAAAA`	`X@A`	282.68	16960	*FuzzBox*
		`X@g>A`	461.45	54963	*Baseline*
tinyexpr_easy	`help(`	`help`	1001.51	5028	*FuzzBox*
		`help`	6.12	339	*Baseline*
tinyexpr_hard	`1+2`	`e+e...(etc)`	11913.52	209312	*FuzzBox*
		-	> 3h	300000+	*Baseline*

obtained by our tool through the QEMU translation process. In a single "*easy*" case, the black box *baseline* approach is quicker than *FuzzBox* since simple mutations suffice to trigger the bug. As *FuzzBox* adopts a gray-box approach, the input generation algorithm focuses on more complex combinations of multiple mutations, to better explore the target code paths. This approach is more effective for hard-to-find bugs and for achieving higher coverage in the long term, even if it is overblown for simpler bugs.

4.4.4 Performance

We assessed the performance of *FuzzBox* and compared it with the *Baseline*, in terms of throughput of fuzzing (i.e., executions of fuzz inputs per second) when applied against MILS-based targets. We separately analyze the three libraries, considering potential variations among the libraries with respect to the time needed to process the inputs. In each experiment, we first submit 100 fuzz inputs to warm up the target, in order to mitigate initial transient variations in the measurements, due to the initialization of the fuzzer. Subsequently, we measured throughput with an additional 1000 fuzz inputs. To ensure statistical significance, we performed three repetitions for each experiment. Table 4.4 provides the average throughput, along with the standard deviation, for both approaches and each target. The results show that, for all three targets, the throughput achieved by

FuzzBox is approximately twice that of the *Baseline*. This underscores that the overhead introduced by *FuzzBox* for profiling, injecting, and instrumenting coverage within the translation process is minimal compared to the overhead associated with a naive fuzzing workflow. Given potential transient slowdowns in the target system (e.g., with large fuzz inputs), we properly set timeouts on the client side to await server liveness messages, long enough to prevent false positives in failure detection. Nevertheless, this configuration significantly affects the overhead on the Baseline fuzzing throughput.

Table 4.4. Throughput of FuzzBox compared to the Baseline.

Target	**Baseline Throughput (#input/sec)**	***FuzzBox* Throughput (#input/sec)**
json	23.05 ± 0.19	47.92 ± 1.72
sendmail	24.65 ± 0.93	53.09 ± 3.53
tinyexpr	20.90 ± 1.55	40.68 ± 0.78

This evaluation highlights *FuzzBox* as a groundbreaking solution for fuzzing industrial embedded systems like MILS-based applications. Unlike the baseline, which relies on custom fuzz drivers and recompilation with limited introspection, *FuzzBox* achieves superior fuzzing depth, discovering hard-to-reach bugs with a 57.37% average coverage improvement. Additionally, it delivers nearly twice the throughput, minimizing the overhead caused by liveness-based crash detection. These results position *FuzzBox* as a high-performance, effective tool for addressing the unique challenges of fuzzing binary-only industrial targets.

4.5 Portability Evaluation

In this section, we evaluate the capability of *FuzzBox* to fuzz targets beyond MILS-based systems. To assess its portability and broader applicability, we applied *FuzzBox* to a diverse set of Linux-based IoT firmware from commercial embedded devices. Additionally, we compared the improvements achieved by *FuzzBox* to a *Baseline* approach based on black-box network fuzzing. While some of the existing state-of-the-art fuzzers

are applicable to IoT systems, the goal of this evaluation is not to compare *FuzzBox* against those tools directly. These fuzzers adopt their own fuzz input generation strategies, which would make it difficult to attribute any observed differences in performance solely to architectural factors. To ensure a fair and controlled comparison, we use a Baseline setup that shares the same LibAFL backend as *FuzzBox*. Since our goal is to validate the general applicability of FuzzBox across different embedded system types, CPU architectures, and operating systems, the Baseline setup assures us that experiments use identical input mutation and scheduling strategies.

4.5.1 Evaluation Targets

These experiments consider Linux-based IoT firmware as targets of evaluation. Unlike MILS systems, IoT firmware runs atop general-purpose Linux operating systems and has been extensively studied as fuzzing targets in the literature [157, 156, 305]. For this evaluation, we selected real-word firmware of three popular IoT devices. We included dual-band gigabit WiFi routers and IP cameras, running on embedded architectures based on ARM and MIPS CPUs. Table 4.5 lists the selected targets, their type, and the CPU architectures that were emulated during the experiments. We executed the firmware on QEMU VMs with 1 GB RAM, running on a host system with an *AMD Ryzen 5 3600 6-Core 3.98 GHz* processor and 12GB RAM.

Table 4.5. Linux-based Firmware for portability evaluation.

Target	Description	Architecture
TENDA AC15 [23]	WiFi router firmware	ARM (little endian)
TEW-651BR [24]	WiFi router firmware	MIPS (big endian)
DCS-932L [21]	IP camera firmware	MIPS (little endian)

4.5.2 Experimental Setup

To evaluate and compare *FuzzBox* coverage-driven approach against traditional network fuzzing approach, we configured two distinct testing setup.

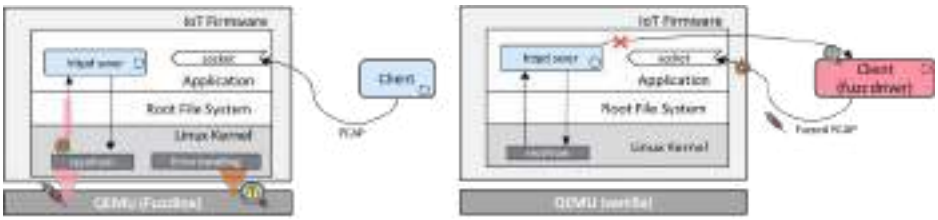


Figure 4.4. *Firmware FuzzBox* (left) and *Firmware baseline* (right) testing setup [62].

FuzzBox Testing Setup. For the evaluation of *FuzzBox* on the IoT firmware target, we setup a Linux-based environment. As shown in Figure 4.4 (left), the Linux-based firmware runs a *httpd* server, actively listening for incoming HTTP requests related to network configuration, security settings, and communication protocols. Upon receiving an HTTP request through the network interface, the firmware utilizes the `recvfrom()` Linux system call to process incoming packets from a socket. In contrast to MILS-based applications, where invocation cycles are periodically triggered, the target firmware requires external stimulation. To achieve this, we included an external client to send a pre-recorded seed in PCAP format as a repeated stream of packets to the server, triggering the functions we aim to intercept and fuzz (i.e., `recvfrom()`). This eliminates the need for a protocol-specific fuzz client, as required in other fuzzing approaches [232].

In order to attack the firmware, we fuzz messages over the socket before they are received by the target application. To enable this, *FuzzBox* intercepts the receiving system call, both before and after its execution (pre- and post-invocation). During pre-invocation, *FuzzBox* collects information about the memory location designated for saving the received messages, and saves the return address of the call. At post-invocation, when the execution reaches the return address, *FuzzBox* performs the actual fuzzing operation on the messages.

Thus, when fuzzing received messages, *FuzzBox* manipulates the buffer of the `recvfrom()` syscall. Since blindly applying mutations across the entire HTTP payload would be inefficient, fuzzing tools for HTTP, such as the Burp Suite [22] and OWASP ZAP [223], are configured to selectively mutate specific fields in HTTP requests, either in the header or body (e.g.,

parameters in the HTTP query string). To address this, *FuzzBox* supports additional configuration parameters, including an offset and the length to fuzz for selective fuzzing of specific bytes within the syscall input/output buffer. This allows users to target specific parts of intercepted HTTP requests for more efficient fuzzing.

Additionally, for Linux-based targets, crash detection in *FuzzBox* is achieved by intercepting Linux core dumps, which are typically generated when a program encounters a segmentation fault or another critical error for the running target application.

In summary, the *FuzzBox* configuration parameters for this setup include setting `recvfrom` as the target syscall, specifying its parameter "*buf*" (at the second position) for fuzzing, and defining the offset and the length within which to fuzz the buffer. The `do_coredump()` signal handler is configured to intercept and detect firmware crashes. Finally, the target CPU architecture is selected, as the parameter to fuzz is passed in different CPU registers depending on the architecture (e.g., the 2nd parameter of a syscall is passed via the GPR4 register in PowerPC CPUs).

Baseline Testing Setup. To provide a basis for comparison with *FuzzBox*, we configured a baseline setup. This configuration, shown in Figure 4.4 (bottom), employs a simple black-box fuzzing approach for IoT firmware. In this setup, a fuzz driver is integrated into the external client, which uses libAFL to generate and schedule fuzzing inputs by mutating the pre-recorded PCAP seed. Users can specify which fields to mutate by applying a mask to selected bytes within the PCAP file. This mechanism enables precise, selective mutation, achieving a granularity level comparable to that of the *FuzzBox* testing setup. The fuzz driver sends these mutated inputs to the target server, which processes them in the same manner as in the *FuzzBox* setup. The fuzz driver then waits for a response from the server, using a timeout mechanism to detect crashes by flagging unresponsive states.

The firmware image, comprising a Linux kernel, root filesystem, and the target application, is executed on a vanilla QEMU instance. Similar to the MILS-based setup, coverage collection of *FuzzBox* is enabled for the evaluation and not utilized as feedback mechanism.

This baseline configuration reflects traditional black-box network fuzz-

ing techniques commonly employed for IoT firmware [305, 65, 107]. In these techniques, the fuzzer operates remotely, communicates with the target through network interfaces, and relies solely on analyzing response messages, without utilizing others feedback mechanisms. However, this approach presents limitations in both performance and effectiveness. First, the timeout-based crash detection mechanism introduces delays, which impact on the fuzzing throughput. Second, the lack of visibility into the target system's internal state prevents the fuzz driver from leveraging feedback mechanisms like edge coverage, potentially reducing its effectiveness in uncovering vulnerabilities that may be hidden deeper in the firmware logic.

4.5.3 Fuzzing Depth

To evaluate the effectiveness of *FuzzBox* against Linux-based firmware compared to traditional network fuzzing, we measured code coverage (edge and basic blocks) achieved during fuzzing and assessed the ability of both setups to rediscover known vulnerabilities. For all evaluation targets in both fuzzing setups, we used a valid PCAP as fuzzing seed, which contains HTTP requests that exercise a known vulnerable functionality of the target firmware. We used the same crafted PCAPs for both *FuzzBox* and the *baseline* setups.

For the Tenda AC15 firmware, we targeted the endpoint "`goform/fast_setting_wifi_set`", which is used for WiFi setup. This endpoint has a known vulnerability in the "`ssid`" field of the HTTP request, which can be exploited to trigger a buffer overflow (CVE-2018-16333) in the router's web server.

For the TEW-651BR firmware, we targeted the `get_set.ccp` endpoint, and we specifically fuzzed the "`ccp_act`" parameter passed through the body of the HTTP request. The manipulation of this parameter can lead to a memory corruption vulnerability (CVE-2019-11400), potentially having impact on confidentiality, integrity and availability.

Finally, for the DCS-932L firmware, we targeted the `wireless.htm` endpoint and the WEPencryption field, leading to a stack-based buffer overflow (CVE-2019-10999) in the camera's web server. This overflow can potentially allow a remotely authenticated attacker to execute arbitrary

code by providing a long string in the "`WEPEncryption`" parameter when requesting wireless.htm.

In summary, *FuzzBox* intercepts and selectively fuzzes bytes in the buffer received by the `recvfrom()` system call while processing incoming HTTP requests. In contrast, the *Baseline* setup mutates bytes in the PCAP before sending (corrupted) requests to the firmware network interface, similarly to existing fuzzing tools for HTTP such as Burp Suite and OWASP ZAP.

Coverage Analysis. We measured the edge and block coverage achieved by *FuzzBox* and the *Baseline* approach on Linux-based firmware. Figure 4.5 illustrates the coverage growth curves over time, demonstrating that *FuzzBox* consistently outperforms the black-box network fuzzing approach in two out of three cases and performs comparably in the third one.

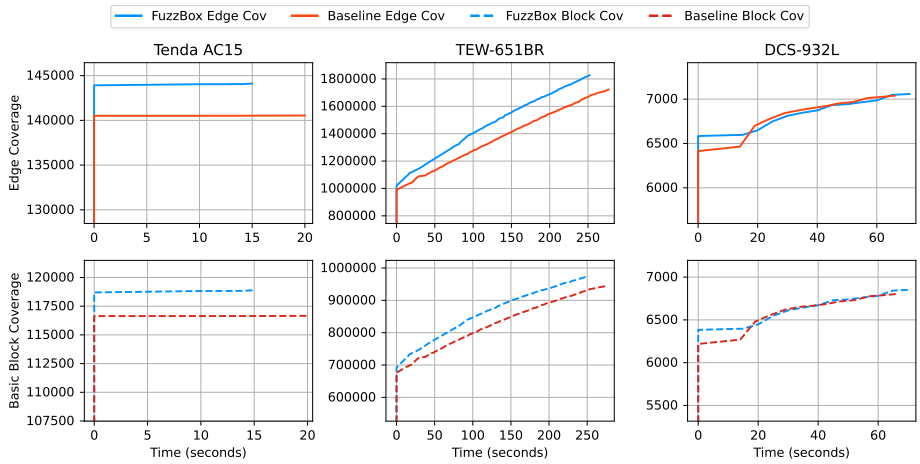


Figure 4.5. Coverage growth curves over time for Linux-based firmware targets [62].

Bug Analysis. In our experiment, we successfully discovered three known vulnerabilities in the target firmware. Leveraging grey-box fuzzing, fewer or same iterations were required to explore deep paths within the target and effectively trigger these vulnerabilities. Table 4.6 summarizes findings on bug discovery, reporting the required iterations to crash, as well as the fuzzing seed, comparing the two approaches.

Table 4.6. Vulnerabilities in *FuzzBox* and *Baseline* for Linux-based firmware.

Target	CVE	Seed	Iter.	Approach
TENDA AC15	CVE-2018-16333	`ssid=seed`	25	*FuzzBox*
			39	*Baseline*
TEW-651BR	CVE-2019-11400	`ccp_act=aaaaaa`	114	*FuzzBox*
			145	*Baseline*
DCS-932L	CVE-2019-10999)	`WEPEncryption=seed`	13	*FuzzBox*
			13	*Baseline*

The evaluation highlights *FuzzBox*'s efficiency and effectiveness in identifying critical security weaknesses in Linux-based IoT firmware, extending its applicability beyond its focus on industrial systems. In this context, unlike traditional network fuzzing, which submits corrupted HTTP requests externally, *FuzzBox* employs on-the-fly corruption of internal IPC interfaces, potentially enabling deeper control flow manipulation. This innovative approach allows it to uncover vulnerabilities that may remain unreachable through network-level fuzzing alone. By achieving higher code coverage and efficiently rediscovering known vulnerabilities, *FuzzBox* demonstrates its portability to diverse embedded environments. While primarily designed for industrial systems, this use case underscores its adaptability to other embedded systems, such as IoT firmware, further validating its versatility.

4.6 Limitations

Despite positive results obtained by using the proposed architecture against MILS-based applications, we briefly discuss limitations and avenues of further improvement.

Emulation Accuracy. Our approach relies on emulation for interception, fuzzing, and feedback. However, there are potential limitations when the target system can not be fully emulated. This might occur due to lacking support for specific peripherals or lower accuracy in replicating real-world interactions, which can impact fuzzing effectiveness in finding security flaws. This could result in false positives or negatives, and limitations in code coverage. To tackle these challenges, we can explore solutions like using the QEMU extensions for automated rehosting of unsupported peripherals [186, 150, 105]. Incorporating Hardware-In-The-Loop (HIL) techniques [86] is another option while keeping our approach for integrating fuzzing with emulation. Despite these limitations, emulation remains a popular choice for assessing embedded systems [269, 113, 306].

Symbol Table Dependency. *FuzzBox* relies on the symbol table to get information about the target function. This table is usually included in the binary during development if debug information is enabled. MILS binaries in ELF format support this. Other binary formats like COFF, PE, and Mach-O, also include it. In production builds, debug information might be stripped to reduce binary size. While it is not the primary purpose of *FuzzBox*, in these use cases, reverse engineering tools like Fuzzable [2] can be used automatically discover fuzzable target functions through static analysis of the binary. Alternatively, advanced techniques such as probabilistic naming of functions can be applied to stripped binaries [231].

Security Mitigations. Certain security measures at the binary level, such as Control Flow Integrity (CFI), Data Execution Prevention (DEP), and stack canaries can co-exist without significantly hindering the *FuzzBox* approach. The primary concern is Address Space Layout Randomization (ASLR), which randomizes the memory layout of the target and introduces unpredictability in system call addresses. While ASLR could be a hurdle, the strategic use of memory forensics tools, such as Volatility [3], [222], emerges as a promising solution for dynamically identifying syscall addresses during runtime. Nevertheless, protections like ASLR are typically enabled for the production release of the target, but not during the testing phase.

Automatic Configuration. While *FuzzBox* currently requires manual effort to configure target functions, parameters to fuzz, and architecture-specific settings (see Section 4.2.4), future work could automate these steps. First, target functions in binaries can be identified using heuristics (e.g., [2], [231]) based on naming conventions, interface signatures, and known APIs (e.g., POSIX or MILS primitives). Dynamic analysis, such as frequent I/O interactions and common call patterns, can refine these candidates. In addition, profiles of known kernels and libraries can be shared for reuse. Second, static analysis tools, such as Ghidra, can help infer pointer arguments and associated size fields to automate parameter selection. Third, although architecture-specific calling conventions must be respected, *FuzzBox* already includes profiles for PowerPC, ARM, and MIPS. Future work could extend support to additional architectures, and analysis of binary metadata (e.g., ELF headers) can guide automatic selection of the architecture profile.

Experimental Validity and Repetitions. The empirical evaluation of *FuzzBox* compares coverage, bug-finding capability, and throughput against a baseline using single executions per experimental configuration. While the fuzzing engine (libAFL/AFL-style) is deterministic given a fixed seed and configuration, fuzzing remains inherently non-deterministic at the system level (e.g., due to scheduling and environmental factors), and repeating experiments would provide a stronger statistical basis for our conclusions. As a result, the reported times-to-crash and coverage values should be interpreted as representative instances rather than aggregated statistics, and the lack of multiple repetitions constitutes a threat to the internal validity of the evaluation. A more extensive study with repeated runs and statistical analysis is an important direction for future work.

4.7 Discussion

Inter-process communication (IPC) mechanisms form a critical yet under-tested attack surface in industrial and safety-critical systems. Their complexity, proprietary nature, and lack of observability make traditional source-based or hardware-assisted fuzzing approaches impractical. This chapter presented *FuzzBox*, an emulation-based fuzzer that overcomes

these barriers by integrating coverage-guided fuzzing within full-system emulation. By leveraging QEMU's dynamic binary translation, *FuzzBox* enables transparent input injection, failure detection, and coverage tracking without requiring source code, instrumentation hooks, or specialized hardware. Experimental results on MILS and IoT firmware targets demonstrate its practicality and portability, achieving significant gains in throughput, coverage, and bug discovery compared to black-box approaches. From a broader perspective, this contribution exemplifies how the principle of attack-surface reduction can be extended to the *communication layer* of software systems. By systematically exercising the behavior of privileged IPC interfaces through adversarial testing, *FuzzBox* reduces the latent exposure of complex multi-partition architectures. The next chapter extends this principle further, shifting from orchestration APIs to hypervisor interfaces, where we explore how virtualization boundaries can be systematically exercised and secured through automated, high-fidelity fuzzing.

Chapter 5

Fuzzing of Hypervisor Surface

5.1 Overview

Hypervisors are widely used across cloud and industrial virtualization stacks, but the privileged boundary between guest and host remains a sensitive attack surface: VM exits move control from the guest to the hypervisor and can expose isolation issues that result in crashes or denial-of-service conditions [207, 206, 205]. Fuzz testing has proven effective at uncovering such problems in complex software systems, including hypervisors [308]. However, existing hypervisor fuzzers targeting hardware-assisted virtualization face three recurring limitations. First, many prior efforts focus on I/O virtualization starting from the same VM state and therefore give limited attention to the vCPU execution paths central to hardware-assisted hypervisors [251, 200, 139, 230, 55]. Second, generating valid and diverse seeds often requires manual effort or bespoke guest setups, which reduces automation and repeatability [111, 299, 251, 252, 32, 115]. Third, producing valid VM states from naive input sequences is difficult: tests that do not respect realistic execution contexts tend to trigger frequent guest crashes and require expensive resets, which degrades fuzzing efficiency and coverage [308].

This chapter introduces *IRIS*[1] , a framework designed to address these limitations by learning realistic VM execution sequences from observed guest activity and reusing them as seeds for hypervisor fuzzing. *IRIS*

[1] Available as open-source at `https://github.com/dessertlab/iris`

records sequences VM inputs during real guest operation (for example, an OS boot), replays those sequences to recreate complex, valid hypervisor states, and then applies fuzz mutations on top of the recreated state. By capturing hardware-assisted execution contexts produced during genuine runs, this approach aims to reduce crash rates during state setup and enable exploration of deeper vCPU behaviors that snapshot-only methods may miss. The current implementation targets the Xen hypervisor in hardware-assisted mode (Intel VT-x).

Our experimental analysis shows that *IRIS* can automatically generate seeds that closely reproduce real guest execution. In our settings *IRIS* achieves code-coverage fitting between 92.1% and 100% compared to the recorded guest runs, and replay times improved between 42.5% and 99.6% relative to the original executions. Finally, a preliminary proof-of-concept fuzzer built using *IRIS* demonstrates the feasibility of the approach by generating valid, diverse VM seeds and exposing additional execution paths in the vCPU subsystem beyond those reachable from handcrafted snapshots.

5.2 IRIS Design

Figure 5.1 depicts the high level architecture of *IRIS*. The framework has been designed regarding Intel VT-x hardware-assisted virtualization technology and it does not strictly depend on the hypervisor under test. In the description of *IRIS*, we use the following terminology: (1) ***VM behavior*** is a sequence $VM\_exit\_trace = \{VMexit_1, ..., VMexit_N\}$ that is the flow of *VM exits* triggered by a workload to reach a valid VM state; (2) ***VM seed*** includes the pairs of VMCS {field, value} read via `VMREAD` instructions, and the values of general-purpose registers (GPR), both obtained during the handling of a VM exit within the *VM_ exit_ trace*.

In the remaining of this chapter, we describe the architectural components that compose *IRIS*, aiming to address the following issues:

1. **Automatic VM seed generation**: reduce the manual effort in VM seed generation, to minimize the required knowledge to develop a fuzzer for hardware-assisted hypervisors;

2. **Accurate VM seeds generation**: generate VM seeds that are accurate in reproducing the real VM behaviors;

3. **Efficient submission of VM seeds**: submit VM seeds in a light-weight fashion, without requiring specific VM guest workload to be executed.

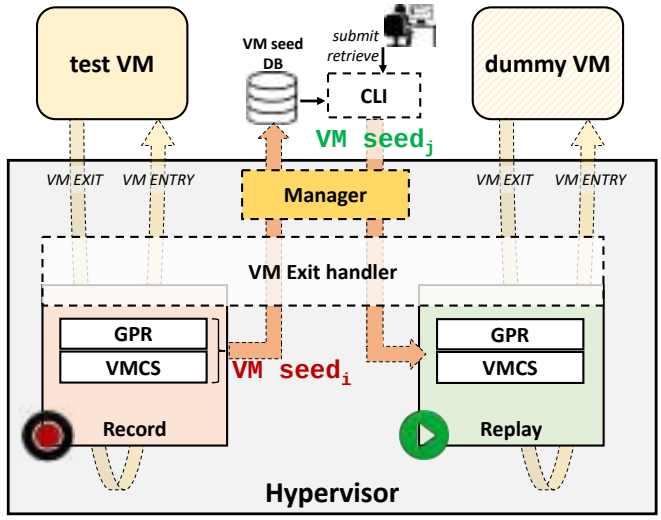


Figure 5.1. Overview of *IRIS* design [59].

5.2.1 Record

The recording component aims to collect a set of information observed while executing the VM, i.e., the *VM behavior*. For each VM exit in a *VM_ exit_ trace*, which is qualified by an *exit reason*, the recording architecture stores *(i)* the *VM seed* and *(ii) metrics*, at hypervisor level.

The *metrics* are necessary to assess the accuracy and efficiency of our framework in replaying the recorded VM behavior. Specifically, the *IRIS* framework records the *i)* code coverage at hypervisor level, *ii)* the pairs of VMCS {field, value} written via `VMWRITE` instructions, and *iii)* and time needed during a VM exit. Code coverage at the hypervisor level is the simplest metric to estimate how much replaying *VM seeds* is *accurate* (refer to subsection 5.2.2, in which we define what is an accurate replay) compared to the recorded VM behavior. However, coverage-related metrics

could not always guarantee to cover critical areas that describe the VM behavior during execution. To mitigate this point, we also included in the *IRIS* framework the monitoring of VMCS field, value pairs accesses, which are peculiar to the hardware-assisted hypervisor solutions. Specifically, we use `VMWRITE`s for a more fine-grained validation of actual VM state changes from the point of view of the VMCS and VMX operations. Finally, the recording component (as well as the replaying, see subsection 5.2.2) gathers time needed to VM exit handling (via *CPU cycle counters*) as an efficiency indicator.

Note that the recording mechanism implemented in *IRIS* allows us to monitor whatever VM exit and related VMCS data/control information (i.e., VMREAD and VMWRITE operations). We deliberately avoid recording the *test VM* memory as the only part of the VM state that we do not monitor. This choice was made to reduce the complexity of VM seeds compared to snapshot-based approaches [115, 251]. In Section 5.4, we demonstrate that this choice is a good compromise to accurately record VM behaviors, and we discuss the few cases in which IRIS can not obtain good accuracy.

5.2.2 Replay

The replaying component allows submitting recorded *VM seeds* to the hypervisor. It is worth noting that the proposed framework also allows submitting crafted VM seeds, i.e., seeds built manually. When a *VM seed* is submitted, the hypervisor executes the related sensitive operations according to the VM exit reason. Note that a sequence of VM exits is usually a result of complex operations that are difficult to be reproduced at the guest level.

Our idea is to reduce the replay time (to improve the efficiency) by enabling a continuous triggering of VM exits that directly stimulate the *VM exit handler*, without actually executing the guest. To perform the replay, we use a *dummy VM* that does not run any operation except *i)* initialization of all hypervisor data structures and VMCS where the seeds will be submitted and *ii)* triggering of VM exits to execute the *VM exit handler* (switch from *non-root mode* to *root mode*).

A possible solution to continuously submit VM exits is to let the dummy VM do a first exit and then implement a loop directly within

the *VM exit handler*. However, a loop in *root mode* could be detected from the hypervisor as a hang, leading to forced crashes. In addition, such a direct loop avoids the VM entry, which occurs as the last step in the VM exit handling. The *VM entry* operation includes several checks on the VMCS fields (specified in Section 26.3 in [141]) that are representative or real VM behavior and are used to guarantee semantically-correct VM seeds submission. Thus, our replaying architecture lets the VM exit handler execute the VM entry and then forces the *dummy VM* to immediately trigger another VM exit, preventing the execution of any instruction at the guest level.

To set up the context of the hypervisor according to the *VM seed*, GPRs are rewritten in the hypervisor data structures where they are stored. The VMCS fields are rewritten (`VMWRITE`s) with *VM seed* values if they are read (`VMREAD`s) again during the replay. If the VMCS fields are read-only, and cannot be rewritten, we modify only the return value of the `VMREAD`s. After the *VM seed* submission, the hypervisor handles sensitive operations according to the *VM seed*, before executing the VM entry operation.

Finally, we define *accuracy* of the replaying mechanism as its ability to reproduce a valid VM behavior for the recorded metrics, i.e., code coverage at the hypervisor level and writes performed into the VMCS fields. To evaluate the accuracy, IRIS allows reverting the *test VM* snapshot saved at the start of recording, and using it as a starting point from which replaying *VM seeds* via the *dummy VM*.

5.2.3 Manager

As already explained, the proposed framework provides the *record* and *replay* operation modes. Further, we implemented a component called *IRIS manager*, which exposes an interface that can be used by a *user-space application (CLI)* to *(i)* choosing between operation modes, i.e., *record* and *replay*; *(ii)* retrieve *VM seeds* and metrics during the *record mode*; *iii)* submitting *VM seeds* during the *replay mode*. When the *IRIS manager* enables the *record mode*, it runs a *test VM* and allows it to trigger normal VM exits as they occur. The *record mode* can be configured to store *VM seeds*, metrics, or both of them. Recording can be manually or programmatically stopped after a given number of monitored VM exits. After a specific time of recording, the *IRIS manager* allows keeping the

test VM in an idle loop, reading for a new recording session; otherwise, the *test VM* continues execution with no recording enabled. In the *replay mode*, *IRIS manager* first runs a *dummy VM* (optionally by reverting to a particular VM state) and then allows users to submit seeds on-demand. Also, in this case, the *manager* puts the *dummy VM* in an idle loop to wait for new *VM seeds* to submit. When a new *VM seed* is available for submission, the replaying component submits it to the hypervisor. Note that both recorded seeds and manually crafted seeds can be submitted at this step. Finally, the *IRIS manager* allows enabling the *replay mode* together with the *record mode* enabled to store metrics while replaying. This latter is necessary to evaluate the accuracy and efficiency of recorded/crafted *VM seeds* which are submitted via the *replay mode*.

5.3 Implementation

The current version of *IRIS* is built upon the Xen hypervisor. The reference CPU architecture is Intel x86 with hardware virtualization extension VT-x. We chose Xen since it is an open-source hypervisor and supports *Hardware Virtual Machine* (*HVM*) mode. HVM mode tries to make full virtualization easier, using the hardware emulation to accelerate CPU virtualization (privileged instructions) and the MMU (page tables). *IRIS* is implemented as a set of patches for the Xen kernel. All *IRIS* components code is written in C language. The details of each component are briefly explained in the following.

Record. Currently, in IRIS we obtain code coverage at the hypervisor level via compile-time instrumentation approaches using *gcov* [16]. The hypervisor codebase should not be instrumented as a whole since we need to avoid most sources of non-determinism, e.g., due to interrupts, kernel threads, and statefulness. We selectively instrument hypervisor components crucial for VM exit handling, such as the abstraction of vCPU, HVM domain-specific functions, and the handler of VMX-related operations. While running, the resulting instrumented binary will write its own basic block coverage to a bitmap, which is exported as a shared memory area accessible at the guest level. We remark that code coverage is cleaned up by removing hits due to the execution of our record and

replay components. Further, code coverage information can be retrieved for each VM seed submitted. Regarding the reads and writes performed into the VMCS fields, the hypervisor uses machine instructions `VMREAD` and `VMWRITE`, wrapped respectively by Xen `_vmread()` and `_vmwrite()` functions. We instrument these functions by adding a callback function invocation to store pairs of VMCS {field, value} read or written in the shared memory area. Regarding the values of guest GPR, they are stored in Xen data structures during VM exit handling, since they are not included in the VMCS. The current implementation, for each VM seed, uses an array of struct to store GPR, VMCS fields read or write. The struct is defined to store: i) a flag (1 byte) that indicates the kind of data; ii) the encoding (1 byte) of GPR (15 values) or VMCS fields (147 values); iii) the value (8 bytes) for GPR or VMCS field. Once again, we invoke a callback function at the start of the VM exit handler execution, to also buffering this kind of data. The temporal metric can be retrieved using instructions to get CPU-cycles counters. For example, Intel provides the `RDTSC` instruction, which reads the current value of the CPU's time-stamp counter.

Replay. The replaying component enables the continuous triggering of VM exits by leveraging a *dummy VM* running in HVM mode. We implemented the VM exit/entry loop described in subsection 5.2.2 by enabling the Intel *VMX-preemption timer* for the *dummy VM*. The *VMX-preemption timer* counts down (from the value loaded by a VM entry) in VMX non-root operation, at a rate proportional to that of the timestamp counter (`TSC`). When the timer counts down to zero, it stops counting down and a VM exit occurs (see Sections 25.2, 25.5.1, 26.6.4 [141]). In our framework, a preemption timer value set equal to zero allows the hypervisor to preempt the *dummy VM* execution before the CPU executes any instructions in the guest.

Regarding *VM seed* submission, we implement callback functions inserted at compile-time and invoked during the VM exit handling to submit GPR and VMCS values according to the *VM seed* values. The GPR values are simply copied to the corresponding hypervisor data structures. The VMCS values can be written into the VMCS by invoking the `_vmwrite()` function. However, this solution is adopted only for writable VMCS fields,

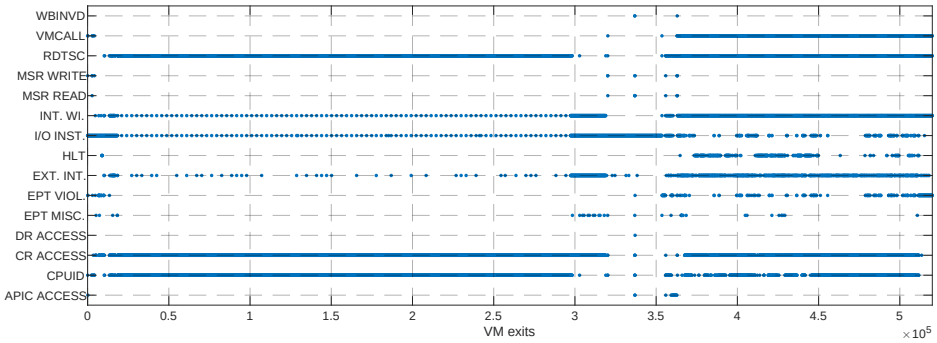


Figure 5.2. VM exit reasons distribution over time during *OS BOOT* workload [59].

since some of them are read-only. For the latter scenario, we instrument the function `_vmread()` by inserting a callback function to replace the values returned from the readings on the VMCS with those submitted with the *VM seed*.

Manager. The manager component consists of a backend driver at the hypervisor level. The interface exposed to users is implemented using the *hypercall* mechanism. We implemented the `xc_vmcs_fuzzing()` hypercall to trap into the hypervisor, to enable and control the recording and replaying phases. Given that, a user-space application (*IRIS* CLI) invokes such hypercall to enable the functionalities provided by *IRIS* manager. Finally, the manager uses `copy_to_guest()` and `copy_from_guest()` Xen routines to respectively retrieve recorded VM seeds and metrics and submit *VM seeds*.

5.4 Evaluation

We perform a thorough experimental analysis of *IRIS*. The aim is to evaluate the accuracy of *IRIS* at reproducing *VM behaviors* using recorded *VM seeds* via the proposed replaying mechanism. To estimate the accuracy, we use the metrics provided by *IRIS* gathered during the execution of guest workloads. Further, we evaluate the efficiency in submitting recorded *VM seeds* via our replaying approach in terms of the CPU time

needed to execute related VM exits. Finally, we provide a proof-of-concept implementation of a fuzzer built upon *IRIS* record and replay mechanisms.

5.4.1 Experimental Setup

We performed our experiments using a host machine with Intel Xeon i7-4790 @3.6Ghz, and 16GB RAM, running Linux kernel v5.10. We implemented *IRIS* on top of Xen hypervisor v4.16, running with *HVM* mode in order to enable hardware-assisted virtualization. Currently, the *IRIS* framework supports Intel VT-x hardware extensions. We run the *IRIS manager* in *Dom0*, and a guest workload to be recorded and replayed in a *DomU*. This latter domain is our *test VM*, while a second *DomU* is mounted as the *dummy VM*. Each domain (both Dom0 and DomU) runs Linux kernel v5.10 and is set up with a single vCPU pinned on a dedicated pCPU, 1 GB RAM, and 20 GB HDD. We impose a 1-to-1 vCPU/pCPU pinning for VMs to prevent as many as possible interferences (e.g., high rate of cache misses) between the different physical cores and obtain coverage data as clean as possible.

5.4.2 Workloads

Experiments are based on workloads that have been characterized by the *IRIS* recording mechanism in terms of *VM behavior*. For the sake of simplicity we consider a sample trace of 5000 VM exits for each workload. In particular, we consider *i)* booting an OS kernel (*OS BOOT*); stressing *ii)* CPU subsystem (*CPU-bound*), *iii)* memory subsystem (*MEM-bound*), and *iv)* I/O subsystem (*I/O-bound*); *v)* keeping idle the OS (*IDLE*).

The *OS BOOT* workload, specifically refers to booting the Linux kernel, and it consists of about 520K VM exits until the OS presents the login screen to the user. Figure 5.2 details the VM exits distribution during *OS BOOT* workload over time. It can be noted that *IRIS* is capable of recording all the VM exits occurring during the boot sequence. The distribution includes a sequence of VM exits (the first 10K) that are related to the BIOS emulated by Xen [296], which is not part of the OS BOOT we want to characterize. Given this, our OS BOOT trace of 5000 VM exits starts after the last BIOS VM exit. The *CPU-bound* workload includes 5000 VM exits triggered during the execution of CPU-intensive operations

(e.g., compute Fibonacci sequence, matrix operations, etc.). The *MEM-bound* workload include 5000 VM exits triggered during the execution of memory-intensive operations for the stack, heap, memory mapping, and shared memory, while *I/O-bound* workload focuses on generic input/output. Finally, the *IDLE* workload includes 5000 VM exits triggered during the OS idle loop.

Figure 5.3 summarizes the distribution of VM exits across the target guest workloads. We can notice that the *OS BOOT* workload results in VM exits that are mostly related to *I/O instruction* and *Control-register accesses* exit reasons. In the boot phase, the guest configures devices and the hypervisor is triggered to carry out operations for emulation, paravirtualization, exclusive assignment, or I/O device sharing depending on the kind of virtualization approach used [27]. In the remaining workloads (i.e., *CPU-bound*, *MEM-bound*, *I/O-bound*, and *IDLE*), almost 80% of VM exits are related to `RDTSC` instructions, which are related to kernel operations needed for timekeeping and implementing scheduling routines [283]. Further, the *IDLE* workload is characterized by some `HLT` VM exits, basically due to the implementation of the *idle loop* in the Linux kernel.

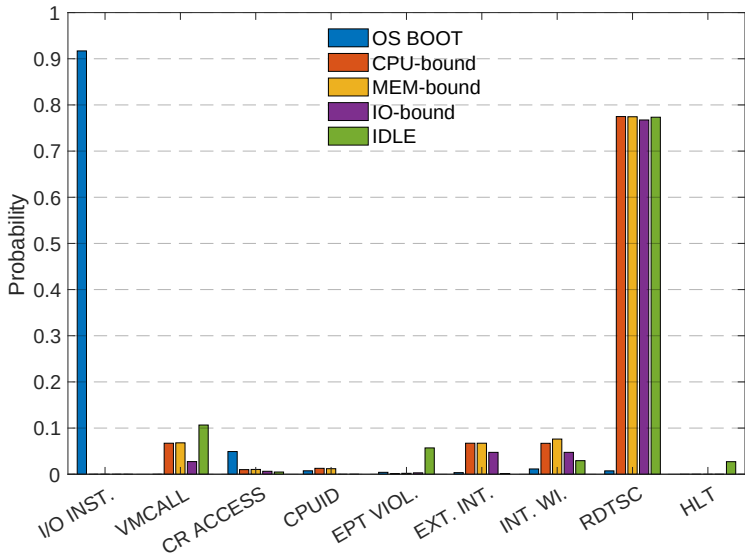
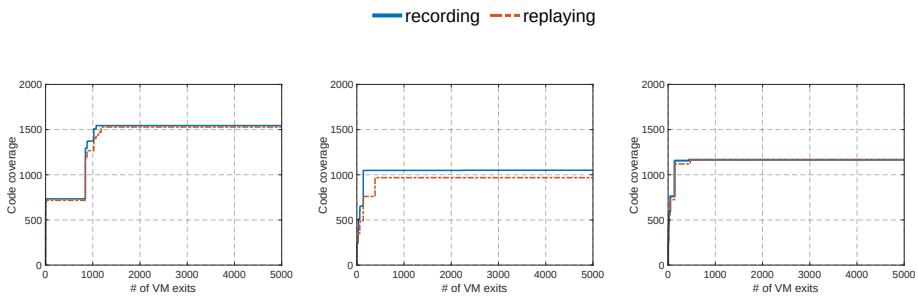


Figure 5.3. VM exit reasons distribution over different target workloads [59].



(a) OS BOOT **(b)** CPU-bound **(c)** IDLE

Figure 5.4. Cumulative code coverage across *OS BOOT*, *CPU-bound*, and *IDLE* workloads [59].

Figure 5.3 also reveals that the hypervisor intervention is relegated to few critical VM exits, which are common across different and heterogeneous workloads. For the sake of representativeness, we further analyzed benchmarks provided in [292], which exhibit similar VM exits distributions obtained above. For simplicity, the analysis provided in the next sections targets only the *CPU-bound* workload since it has several commonalities with the MEM-bound and I/O-bound experimented workloads.

5.4.3 Accuracy

We analyze how accurate is *IRIS* in the automatic recording and replaying of *VM behaviors* using key metrics mentioned in Section 5.2, i.e., code coverage and the pair {VMCS field, value} written. For this purpose, we aim to reveal the difference between metrics gathered both in the recording and replaying phases for each VM seed obtained during the execution of target workloads described in Section 5.4.2. Note that we use the same VM snapshot as the starting state for the record and replay phases to unbias the accuracy evaluation.

Regarding accuracy in terms of code coverage, Figure 5.4 shows the results across the target workloads. The recording curve identifies the cumulative coverage trend for *VM seeds* recorded, in which we evaluate the unique lines of code discovered during VM exit handling, for each *VM seed*. Instead, the replaying curve identifies the cumulative coverage trend

for the replaying of the same VM seeds. The code coverage fitting at the end of replaying *VM seeds* is 99.9%, 92.1%, *and* 98.9% for *OS BOOT*, *CPU-bound*, and *IDLE* workloads, respectively.

Despite we achieved high accuracy in code coverage, we further analyze the remaining differences qualitatively. Figure 5.5 shows the code coverage differences across the three target workloads, which we clustered by VM exit reasons. The code coverage data may be affected by sources of non-determinism due to asynchronous events that interrupt the hypervisor (in VMX root mode) during the VM exits handling. The minimum code coverage difference ranges from 1 to 30 LOC. By analyzing such cases, we point out that such differences are related to the *local Advanced Programmable Interrupt Controller ("vlapic.c")*, *interrupt handling* (*"irq.c"*), and *virtual timer* (`vpt.c`) Xen components. We can treat such coverage differences as noise to filter out.

We also investigate the cases when the code coverage differences are greater than 30 LOC. The frequency of these cases (filtering the repeated VM seeds in a workload) is 0.36%, 0.18%, and 1.16% for *OS BOOT*, *CPU-bound*, *IDLE* workloads, respectively. These differences refer to the *HVM instruction emulator* (`"emulate.c"`) and *VM exit handler* (`"intr.c"`, and `"vmx.c"`) Xen components. This behavior can be due to the recorded VM seeds that are linked to memory-related VM exits. For example, VMCS fields like *Global* and *Local Descriptor Table Registers* (GDTR and LDTR) include references to the memory of "exited" guest VM. Such values can be dereferenced by the hypervisor during exit handling.

We further evaluate the IRIS accuracy, by focusing on the `VMWRITE`s metric, which provides a more fine-grained measure of actual VM state changes (guest-state area) from the point of view of the VMCS and VMX operations.

Focusing on the entire *OS BOOT* workload (see Figure 5.2), the OS switches operating modes and CPU states several times. In that case, the fitting on the executed `VMWRITE`s on the VMCS guest-state area is 100%. Indeed, Figure 5.6 shows an example for `VMWRITE` operations both recorded and replayed against the control register zero (i.e., `CR0`). Each of the modes represents a set of states held by the CR0 register.

In particular, *Mode1* and *Mode2* indicate *real mode* and *protected mode*, respectively. *Mode3* specifies *protected mode* with *paging enabled*,

Mode4 includes *Mode3* with alignment checking performed, *Mode5* includes *Mode4* with test of task switch flag, *Mode6* includes *Mode4* and caching enabled, *Mode7* includes *Mode5* and caching disabled. Results suggest that the proposed recording approach allows us to generate seeds that closely follow real *VM behaviors* of guest execution. Finally, we run an experiment to provide evidence that replaying recorded VM seeds allows reaching the same hypervisor state as in the real guest execution. We replay *CPU-bound* and *IDLE* workloads from a *i)* VM state without booting the OS, and from a *ii)* VM state reached by replaying the recorded *OS BOOT* VM seeds. In the former case, the *dummy VM* crashes (Xen logs: `bad RIP for mode 0`, where `mode 0` is *Mode1* in Figure 5.6), while in the latter case both the *CPU-bound* and *IDLE* workloads complete.

5.4.4 Efficiency

Seeds submission is fundamental in developing fuzzers since it heavily impacts fuzzing efficiency [308]. To this end, we estimate how *IRIS* is

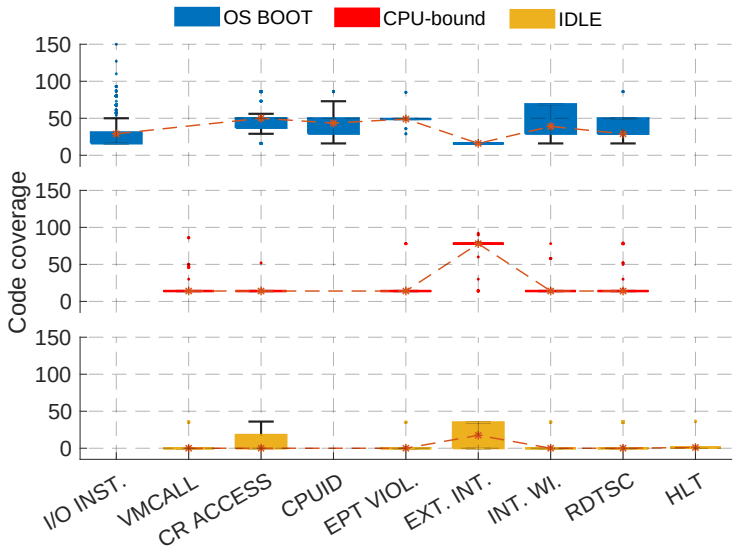


Figure 5.5. Code coverage differences by VM exit reason across targeted workloads [59].

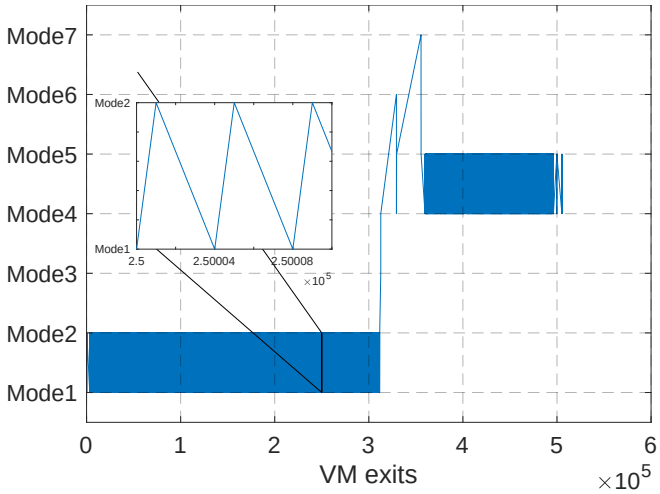
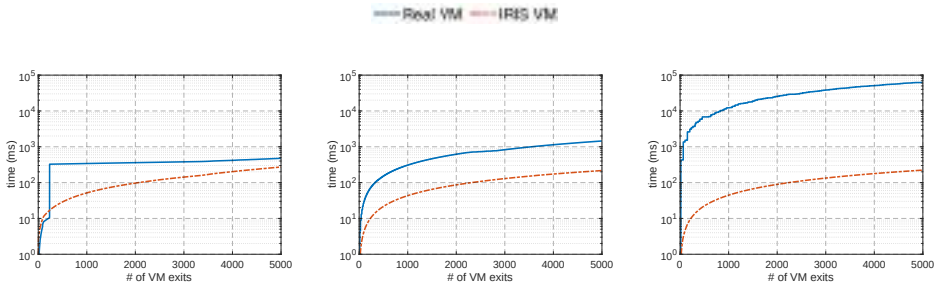


Figure 5.6. Operating modes and virtual CPU states across VM exits during *OS BOOT* workload [59].

efficient in reaching VM states compared to the real guest VM execution. We performed this analysis across workloads described in subsection 5.4.2, by running the experiments 15 times for statistical significance purposes, obtaining the same results with a high level of confidence (p-value < 0.05).

Figure 5.7 shows the time needed to submit VM seeds by real guest VM execution (see *Real VM* in Figure 5.7 and by using the IRIS replaying mechanism (see *IRIS VM* in Figure 5.7). The results show that our replaying mechanism can replay real guest VM behaviors efficiently, with a percentage decrease of 42.5% ($0.27s$ vs $0.47s$), 85.4% ($0.21s$ vs $1.44s$), and 99.6% ($0.22s$ vs $62.61s$), for *OS BOOT*, *CPU-bound*, and *IDLE* workloads respectively. It is worth noting that the *OS BOOT* exhibits the main differences in the first 1000 VM exits, in which the kernel OS spends time running guest operations that do not require the hypervisor intervention. These non-sensitive instructions delay substantially the subsequent VM exits that eventually need to be handled by the hypervisor; thus, there is a non-negligible latency in discovering the same coverage compared to the replayed workload. In general, the throughput of our replaying mechanism is roughly linear, as also confirmed by the time distributions to replay *CPU-bound*, and *IDLE* workloads (see Figure 5.7b and Figure 5.7c). These



(a) OS BOOT **(b)** CPU-bound **(c)** IDLE

Figure 5.7. Performance in submitting *VM seeds* across *OS BOOT*, *CPU-bound*, and *IDLE* workloads [59].

workloads require less hypervisor intervention (VM exits handling), thus the IRIS replaying is even better to discover the same coverage as real guest execution in less time, with a speedup factor of 6.8× and 294× for *CPU-bound* and *IDLE* workloads, respectively.

In addition, we also measure an *ideal replaying throughput* to have an upper bound for estimating the maximum replaying efficiency. We computed this value by running the preemption timer VM exits in the same number of the VM exits needed per workload (i.e., 5000 VM exits), and measure the time needed to handle them. We obtained $0.1s$ ($\sim 350M$ CPU cycles for our testbed), which means $50K$ VM exits/s. Comparing to this *ideal replaying throughput*, we obtained a percentage difference of 63% (18.518 VM exit/s), 52% (23.809 VM exits/s), and 55% (22.727 VM exits/s) for *OS BOOT*, *CPU-bound*, and *IDLE* workloads, respectively. However, such difference does not contemplate the logic behind replaying mechanisms, but it is useful to make new room for improvements.

5.4.5 Performance Overhead

About the overhead induced by the IRIS recording process, we first analyze the target workloads (i.e., *OS BOOT*, *CPU-bound*, and *IDLE*) with the aim to reveal the temporal overhead for each VM exit. We run the workloads 10 times, taking the median values of time needed by the Xen *VM exit handler* to serve a specific VM exit. Figure 5.8 shows the boxplots across the VM exits handled during workload execution, with

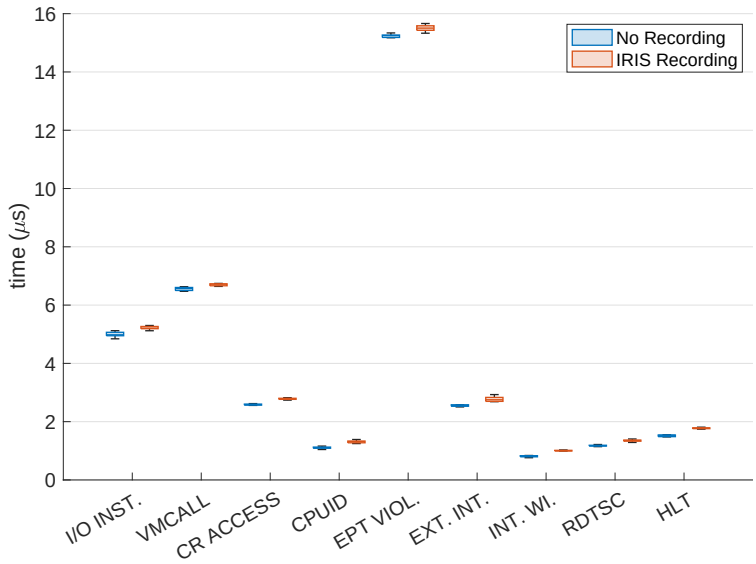


Figure 5.8. The temporal overhead, for each *VM exit*, induced by IRIS recording [59].

and without IRIS recording activated. The results show a very small overhead, ranging from $1,02\%$ to $1,25\%$ percentage increases in the best and worst cases, respectively. Concerning the IRIS memory overhead induced during recording/replaying, we need to consider the size of the *VM seed* for each VM exit and the reads/writes performed on the VMCS. In the worst case, we experimented 32 VMREAD/VMWRITE operations on the VMCS across all the target workloads, obtaining a VM seed size of 470 bytes for each VM exit. The current implementation of the IRIS recording pre-allocates a heap memory equal to the VM seed size in the worst case (i.e., 470 bytes) for each VM exit to be recorded. Instead, the IRIS replaying allocates exactly the needed heap memory for each VM seed recorded since we know in advance the number of VMREAD/VMWRITE operations performed on the VMCS.

5.4.6 IRIS-Based Fuzzer Prototype

We build a proof-of-concept (PoC) to show the potential of using *IRIS* to effortlessly run fuzzing experiments on Xen, as an example of a hardware-assisted solution. The aim is to show that the PoC fuzzer can discover new code coverage and detect anomalous hypervisor behaviors. The fuzzing logic includes *i)* adopting the IRIS replay mechanism to move into valid VM states by utilizing *VM seeds* obtained during the recording of target workloads (i.e., *OS BOOT*, *CPU-bound* and *IDLE*), and ii) mutating a specific VM seed by corrupting VMCS fields and GPR. According to the hypervisor fuzzing literature, we consider the guest VM untrusted. Specifically, the guest VM operations affect directly the VMCS (guest state) and indirectly the hypervisor control flow.

Test cases. The test cases we plan are characterized by the following factors: *i)* the replayed *VM behavior* W of target workloads, *ii)* a target *VM seed* $VMseed_R$ took randomly within the *VM behavior*, and *iii)* the *VM seed* area $A = \{VMCS, GPR\}$, of $VMseed_R$, to mutate.

Each test case starts from an initial VM state s_0 of W (i.e., by starting the VM). Next, the fuzzing logic uses IRIS to replay the *VM behavior* until $VMseed_R$ is reached, to move to the linked VM state (see s_1 in Figure 5.9). At this point, the fuzzer mutates the $VMseed_R$ by generating M (set equal to 10000) mutated versions, defining the *fuzzing sequence* as $C(VMseed_R)_1, ..., C(VMseed_R)_M$, which moves the hypervisor into an unseen state s_M. Such a sequence can be submitted via the IRIS replay mechanism, according to chosen *mutation rules*, as depicted in Figure 5.9.

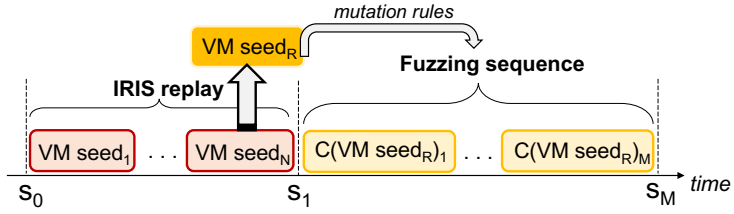


Figure 5.9. Test cases structure in the IRIS-based fuzzer prototype [59].

Table 5.1. New code coverage discovered across test cases by using IRIS-based fuzzer prototype.

Exit Reason	OS BOOT		CPU-bound		IDLE	
	VMCS	GPR	VMCS	GPR	VMCS	GPR
EXT. INT.	+122%	+76%	+7%	+3%	+7%	+3%
INT.WI.	+115%	+61%	+6%	+3%	+6%	+3%
CPUID	+124%	+71%	+14%	+2%	–	–
HLT	–	–	–	–	+7%	+2%
RDTSC	+120%	+69%	+17%	+2%	+17%	+2%
VMCALL	–	–	+18%	+3%	+16%	+3%
CR ACC.	+10%	+63%	+13%	+2%	+10%	+2%
I/O INST.	+8%	+58%	–	–	–	–
EPT VIOL.	+22%	+50%	+13%	+1%	+18%	+10%

Mutation rules. The structure of test cases described above includes the *fuzzing sequence* to be submitted. These mutations focus on a specific *VM seed* area (i.e., VMCS or GPR) of the $VMseed_R$. The mutation rule we adopt includes a *single bit-flip* in *VM seed* area. Specifically, the fuzzer randomly selects a VMCS field or a general-purpose register and then bit-flip the value (e.g., `0xFFFFFFF0` to `0xFFFFFFF1`).

Failure modes. By using scripts that analyze hypervisor behavior and logs, the PoC fuzzer can detect failures occurring during the execution of test cases, that we classify as hypervisor or VM crashes. These can be due to double faults, an invalid operation, page faults, etc. In these cases, the test case, as well as the submitted VM seeds, are saved for further investigation with the aim of crash analysis to reveal potential bugs in the source code.

Results. Table 5.1 shows the new code coverage discovered by running planned test cases, as explained earlier. The code coverage we consider as the baseline is discovered by the single $VMseed_R$, while each cell (i.e., a test case) of Table 5.1 shows the percentage increase of code coverage discovered by submitting the *fuzzing sequence*. In all tests, we can observe newly discovered coverage, with a significant increase in the *OS BOOT* case, due to the complexity of the workload itself. Note that code coverage information can be retrieved for each VM seed submitted. Regarding

failures, we observed VM or hypervisor crashes in respectively 1% and 15% of the tests when the VMCS is mutated. A small number of VM crashes has also been observed when mutating the GPR together with a `CR ACCESS` (as exit reason). In all other cases, the hypervisor is not affected by the mutation. The results show how the IRIS-based fuzzer PoC can discover new hypervisor code coverage and crashes, by planning a few test cases with a naive mutation rule. The manual effort in building fuzzing seeds is negligible since we obtained them by leveraging the IRIS recording mechanism. Furthermore, seed submission is done by reusing the IRIS replay mechanism with no other external tools.

5.5 Limitations

Despite positive results obtained by using the proposed record and replay framework, we briefly discuss limitations and avenues of further improvement.

Memory-related. VM seeds effectiveness. *IRIS* framework does not replay accurately some *VM behaviors* as discussed in the subsection 5.2.2. Specifically, the recording mechanism deliberately does not store guest VM memory areas touched during VM exit handling. These areas can be related to Memory-mapped IO (MMIO) and Port-mapped IO (PMIO) operations between the guest VM and virtualized devices. The behavior of some virtualized devices would be neglected by the current version of IRIS when they use DMA areas, which are beyond the VM exit mechanism. We plan to explore a way to both record efficiently guest VM memory areas and replay accurate the recorded VM seeds, in terms of code coverage. For example, we can exploit the Extended Page Table (EPT) [141] approach provided by Intel processors to virtualize the memory resource in hardware-assisted virtualization. We could only record accessed memory areas during the workload execution at the hypervisor level, and link them to the information already gathered by the *IRIS* framework with the VMCS. Moreover, we can exploit dedicated utility functions implemented by a hypervisor, to read and write guest memory areas. For example, the Xen `hvm_copy_from_guest()`/`hvm_copy_to_guest()` routines are used in the `copy_from_user_hvm()`/`copy_to_user_hvm()` core routines as guest

memory accessors. Anyway, this software-based mechanism would be less performing compared to the EPT mechanism, which is hardware-based.

Replaying efficiency. According to the evaluation part, *IRIS* replays *VM seeds* with an efficiency that settles around half of the ideal replaying throughput of 50000 VM exits/s in our testbed. However, at the current state, *IRIS* does not include any form of optimization for *VM seeds* submission. In fact, VM seeds are submitted one by one; the replay mechanism consumes one *VM seed* and waits for the next. Typically, a fuzzer, generates a large set of fuzzing inputs (i.e. *VM seeds* in our case), applying concurrently multiple mutations to the initial seed. Submitting a *VM seeds* in batch, or implementing buffering mechanisms to continuously submit *VM seeds* as they are generated, could increase the overall replay throughput. In the next releases of *IRIS*, we plan to implement these architectural optimizations.

Code coverage. The code coverage is a paramount metric to guide a fuzzer in interesting points of target source code. The current implementation of *IRIS* leverages a software-based approach like *gcov* [16] to retrieve such code coverage data independently of the CPU architecture. Other hardware-based mechanisms, like *Intel Processor Trace* (Intel PT) [141], allows recording complete control flow with low-performance overhead while not modifying the target hypervisor. We plan to experiment *Intel PT* (see Chap. 35 in [141]) in *IRIS* to make feasible an efficient coverage-guided fuzzer. However, Intel PT can not be used for a hypervisor that does not target Intel VT-x extensions. Also, we plan to use *gcov* `flush()` routine to push the coverage data periodically with no need of manual retrieving. Besides, code coverage should be accompanied by other metrics to cover critical areas that describe the VM (mis)behavior during execution. To mitigate this point, IRIS allows monitoring of VMCS {field, value} pairs, which are peculiar to the hardware-assisted hypervisor solutions. However, it is necessary to deepen the understanding of which other relevant metrics could be used to enable the proper assessment of hardware-assisted virtualization.

Fuzzing. In this study, we only provided a proof-of-concept fuzzer based upon a record/replay framework, as an example of an assessment solution that addresses hardware-assisted virtualization. However, the simpler mutation rules adopted do not cover the complex fuzzing logic that is adopted by current state-of-the-art fuzzers. We plan to perform a thorough fuzzing experimentation, exploiting the findings provided in this study, to develop a fuzzer aimed at discovering vulnerabilities for hardware-assisted hypervisors.

Portability. *IRIS* framework is currently implemented for Xen hypervisor and Intel VT-x virtualization extensions. The proposed design can be easily ported to hypervisors that support Intel VT-x (e.g., KVM [159]) since the logic of VM exit handling is well-defined. Regarding the target CPU architecture, different vendors provide their own virtualization extensions. AMD SVM [31] defines the *Virtual Memory Control Block* (VMCB) data structure, which holds information for the hypervisor and the guest similarly to the VMCS. AMV SVM introduces the *world switch* to indicate the context changes between the hypervisor and guests, and the VMCB for a guest has settings that determine what actions cause the guest to exit to host. ARM virtualization extensions (VHE) [81] are centered around privilege levels, also called *exception levels*. ARM includes running user and kernel at EL0 and EL1 levels, respectively, and adds a new CPU privilege level (EL2) to run hypervisor code. A hypervisor running in EL2 can configure the hardware to support VMs, which run in EL0 and EL1 levels. VMs execute normally until some operation (e.g., sensitive instructions) requires the intervention of the hypervisor, leading to traps into EL2 giving control to the hypervisor, and implementing a classic context switch between processes. We plan to address the peculiarities of the most popular CPU virtualization extensions in the next releases of *IRIS*.

Multiprocessor and multicore support. Multiprocessor support provided by a hypervisor can be a source of non-determinism and security-related concerns. Accordingly, the *IRIS* framework should differently treat Symmetric Multiprocessing (SMP) and Asymmetric Multiprocessing (AMP) architectures, since the hypervisor code, and consequently, the VM exit handler code could be common to all the physical processors or not. At

the time of writing, we performed experiments only by focusing on the Xen hypervisor, which currently supports only SMP architecture for the Intel processors. We plan to explore AMP support of popular hypervisors, like the Xen version for ARM-based platforms. Regarding the issue of having multicore-powered VMs, the current version of IRIS can record and replay VM behaviors according to the VMCS structure provided by Intel VT-x, which is created for each virtual CPU. Thus, the *IRIS* framework can record/replay different flows of vCPU behaviors in the same VM. For example, the Xen hypervisor leverages a structure named `struct vcpu` that keeps track of the current vCPU exited towards the hypervisor.

5.6 Discussion

This chapter addressed the challenge outlined in Section 2.2.3: the absence of scalable methods to exercise the full hypervisor surface under hardware-assisted virtualization. We presented *IRIS*, a framework that records and replays *VM behaviors* from real guest executions to enable effective fuzzing. By leveraging VM exit transitions, the core abstraction of hardware-assisted virtualization, *IRIS* captures valid guest–hypervisor interaction sequences and reuses them as VM seeds, enabling deterministic and efficient replay of complex states without manual effort or instrumentation. This approach allows systematic exploration of deep execution paths across CPU and device virtualization components. Our evaluation shows that *IRIS* accurately reproduces real execution behavior, with code coverage closely matching that of the original guest and replaying tens of thousands of VM seeds per second, thus greatly reducing the manual and computational overhead of hypervisor fuzzing. A proof-of-concept fuzzer built on *IRIS* further demonstrates its practicality, confirming that replayed VM states serve as effective fuzzing seeds for testing privileged hypervisor paths. Overall, *IRIS* represents a key step toward automated, high-fidelity assessment of hardware-assisted virtualization and provides a strong foundation for future research on hypervisor isolation and security. The next chapters extend this progression, shifting from testing system-level interfaces to securing the higher-level interactions that shape software construction and distribution within the supply chain.

Chapter 6

Identifying Software Supply Chain Attack Vectors

6.1 Overview

Modern software development increasingly depends on OSS dependencies, enabling rapid reuse but introducing serious supply chain risks. A single compromised package can cascade through thousands of downstream applications. Recent years have witnessed an alarming rise in such attacks [80], including typo-squatting campaigns on npm [175], crypto-mining malware on PyPI [266], and large-scale repo-jacking incidents affecting over 15,000 Go repositories [43]. The Go ecosystem, used in critical software such as Kubernetes and Go Ethereum, is equally exposed. For instance, a typosquatted Go package named `cli` stealthily exfiltrated sensitive user data through a malicious initialization function. Despite the growing prevalence of these threats, existing taxonomies [218, 176, 177] of supply chain attacks remain largely language-agnostic, overlooking Go's unique dependency model and compilation pipeline. Similarly, current Go security tools [226, 227, 225, 224, 124] address only narrow subsets of possible attacks, failing to capture stealthy behaviors introduced through dependency abuse or malicious initialization code.

This chapter introduces *GoSurf*[1], the first systematic framework to characterize and detect malicious code concealment in the Go ecosystem.

[1] Available as open-source at `https://github.com/chains-project/GoSurf`

GoSurf builds on a novel taxonomy defining 12 distinct attack vectors specific to Go's language features and dependency structure, covering diverse mechanisms for stealthy propagation. We make this taxonomy actionable through a static analyzer that scans Go modules for these vectors, quantifies their occurrence, and tracks their evolution across versions.

Applying *GoSurf* to a dataset of 500 popular Go modules reveals that all twelve attack vectors occur in the wild, with many projects exhibiting broad and persistent attack surfaces. Moreover, longitudinal analysis across module versions highlights how attack exposure evolves over time.

6.2 Supply Chain Attack Vectors in Go

A supply chain attack typically involves two main stages: first, an attacker introduces malicious code in an upstream software component, which subsequently is propagated to downstream users. The second stage is the execution of this malicious code. In this chapter, we focus on supply chain attacks related to source code, and define 'attack vector' as follows.

> An attack vector is a language feature that can potentially be exploited by an attacker to hide and execute malicious code during the application lifecycle.

These source code attack vectors are highly programming language specific. We note that the usage of these features is not inherently malicious, and they are indeed mostly used for good. Yet, we claim that, as potential attack vectors, they should be prioritized for security audits, as they are more prone to hiding the execution of malicious code. Establishing a robust taxonomy of these attack vectors is critical for comprehending and mitigating software supply chain risks. Although previous efforts [177, 218] have yielded general taxonomies for supply chain attack strategies for arbitrary code execution, they are not tailored to the unique characteristics and idiosyncrasies of specific programming languages and stacks.

In this section, we propose a novel language-specific taxonomy to classify attack vectors for code execution in Go packages. With this taxonomy, we can help researchers and security experts to effectively understand and identify the unique risks linked to Go dependencies. To develop a Go-specific taxonomy of supply chain attack vectors, we began by collecting

two existing Go-specific vectors from the literature [177] (constructors and init functions). Next, we reviewed known attack vectors in other languages [218] and their relevance in Go, identifying an additional applicable vector (testing functions). Finally, by analyzing the Go documentation, we identified nine other vectors relying on Go-specific features, not covered by any previous work. To validate these findings, we curated a dataset of executable proof-of-concept attacks written in Go, included in the project repository.

We categorized the identified attack vectors according to three phases of the package lifecycle: pre-build, initialization, and execution (see Sec. 2.1.4 for a detailed explanation of these phases). For each of the identified attack vectors, we provide a detailed explanation of the Go-specific feature and its intended purpose, as well as how it can be exploited to perform an attack on the supply chain.

6.2.1 Malicious Code at Pre-Build Time

During the pre-build phase, third-party dependency integration encompasses various operations. The Go toolchain provides two primary tools to streamline these operations: `'go generate'` for automating code generation tasks, and `'go test'` for testing the code. Attackers can exploit both operations to hide and execute malicious code.

P1. Static Code Generation

The `go generate` tool enables developers to define and execute *code generators*, which run commands and generate Go source code. Go generators are defined in comments using the `'//go:generate'` directive, followed by a shell command to be executed, e.g. `'//go:generate cmd'`. Then, when the `'go generate'` tool is run, source files in the package are scanned for these directives, executing the specified shell commands. It is important to note that `'go generate'` is not part of the `'go build'` command by default and must be explicitly run before building the project [234].

Attack This feature poses two main security concerns. First, generators can execute arbitrary shell commands, allowing attackers to insert malicious code into the directive to be directly executed. Second, generators

can generate additional source code on the fly to attach to a Go project before building, thereby enabling the injection of malicious code into the final executable during the subsequent build. Malicious code hidden in these directives can be unintentionally executed by a developer. This can happen when imported dependencies include code generation operations that need to be run before compilation. In addition, CI/CD pipelines might automatically run the
'go generate' command when invoked from a build script. For example, as evidenced by real-life Go projects such as Istio [147], Terraform [133], and Vault [135], it is a common practice to run code generation in Makefile scripts. A real-world example [120] of this pattern is shown in Listing 6.1. Although the directive does not directly execute any harmful commands, the code fetched from an external source can be inherently malicious.

Listing 6.1. Usage of the `go:generate` directive.

```
1 //go:generate bash -c "mkdir -p codegen && go run github.com/
    deepmap/oapi-codegen/cmd/oapi-codegen@v1.12.4 -generate
    types,server,spec -package codegen api/casaos/openapi.yaml
     > codegen/casaos_api.go"
```

P2. Testing Functions

The Go Toolchain includes built-in features to facilitate robust and efficient software testing. Two central components of the Go testing framework are the 'go test' tool and the standard library `testing`, which automates the execution of test cases. Developers can create test files ending in '_test.go', and define functions starting with the prefix 'Example', 'Test', 'Benchmark' or 'Fuzz' to design examples, tests, benchmarks, and fuzzing, respectively. These functions are automatically executed by the 'go test' tool. This feature is intended to be used during the testing stage of development or as part of the CI actions. Dependency consumers may also use it to test imported code before integration and compilation.

Attack Test suites create a low-visibility environment with potentially large amounts of code, allowing attackers to hide malicious code for seamless execution during build integration. This code might compromise either the developer machine or the CI runners. Testing functions may be over-

looked in security reviews because of the absence of business logic that ends up in production. However, some testing functions can be triggered during post-deployment smoke tests. This provides attackers with the opportunity to execute arbitrary code on production machines.

6.2.2 Malicious Code at Initialization Time

As mentioned in Sec. 2.1.4, the Go Runtime supports initialization operations executed before the main logic by means of *global variable initialization* and `'init'` functions. Although these mechanisms are useful for initializing packages, arbitrary code can be injected and executed through them.

I1. Global Variable Initialization

In Go, any variable declared outside a function is referred to as a *package-level global variable*. Global variables can be initialized using simple expressions, such as a composite literal, or when more complex operations are required by the return value of regular or anonymous function calls. From a runtime perspective, functions (regular or anonymous) invoked on the right-hand side of a global variable declaration statement, are executed promptly during package initialization, before the importing program executes both the `'main'` and any `'init'` functions.

Attack The initialization logic creates a stealthy attack vector. Because global variable initializations are executed before the `'main'` function, an attacker can establish a foothold and potentially manipulate the application's behavior from the outset. Furthermore, this initialization logic is executed silently whenever a compromised package is imported, either directly or transitively, through nested dependencies, even if the package is not used later. Examples of both regular and anonymous functions used for global variable initialization are presented in Listing 6.2.

Listing 6.2. Global variable initialization with function calls.

```
1 func UnsafeMethod() string {
2     // malicious code
3 }
4 var var1 string = unsafeMethod()
```

```
var var2 string = func() string {
    //malicious code
}()
```

I2. Initialization Hooks

Initialization tasks can also be defined in `'init()'` functions. After initializing all global variables in a package, the Go runtime runs the `'init()'` functions of dependencies.

Attack As for global variable initialization, an attacker can inject malicious code into an `'init'` function to execute their payload automatically during the import process, before the main program starts. Note that Go removes unused dependencies such that if a package is imported but never directly used, its `'init()'` function will not be executed. However, when importing a package with an underscore prefix, Go prevents the dependency from being removed, which ensures its `'init'` function always runs [274].

6.2.3 Malicious Code at Execution Time

In Go, certain constructs can mask and execute harmful code during the main execution phase. These include hidden dynamic behavior and unsafe features.

E1. Constructor Methods

In Go, developers define custom functions that serve as constructors for structures following a naming convention such as `'NewStructName'` [279]. If the variable being initialized is the primary type for the given package, the constructor can be named `'New'` without suffix.

Attack Constructors are called to create a new instance of a structure, and any malicious code embedded into constructors will execute during each instantiation. Since constructors are often expected to only handle simple initialization tasks, they may be overlooked as potential vectors for

malicious activity. This makes hiding malicious code in constructors an effective method for attackers to compromise a system repeatedly.

E2. Reflection

Reflection in Go enables dynamic inspection and runtime modification of types and methods. This is accomplished through the `reflect` package. When a method is invoked through reflection, the actual method called is determined at runtime rather than at compile-time. Developers use reflection to build generic libraries, implement serialization and deserialization mechanisms, and create flexible APIs that can operate on various types without needing to know their specifics at compile-time [282].

Attack An attacker can exploit reflection to dynamically inject and execute arbitrary code at runtime. For instance, consider the scenario shown in Listing 6.3; reflection is used to call `'UnsafeMethod'` dynamically. The method name to be called is stored in a variable that an attacker can manipulate (lines 10-11) to execute arbitrary methods. This ability to invoke methods dynamically makes it easier for attackers to hide malicious payloads and evade static security analysis [250].

Listing 6.3. Indirect method invocation through reflection.

```
1  type Mytype string
2  func (t MyType) Safe() {
3      // benign code
4  }
5  func (t MyType) Unsafe() {
6      // malicious code
7  }
8  func main() {
9      var target MyType
10     var methodName string = "Safe"
11     methodName = "Unsafe" // hidden manipulation
12     v := reflect.ValueOf(target)
13     m := v.MethodByName(methodName)
14     m.Call(nil)
```

E3. Interface Polymorphism

In Go, polymorphism is achieved via *interfaces*, which specify a set of method signatures, i.e., behaviors. Interfaces in Go are structurally typed rather than nominally typed. Any type is considered to implement an interface if it is structurally equivalent to the interface (i.e., implements all its methods) regardless of the explicit declaration. When an interface is used, the actual invoked method is dynamically dispatched based on the method set of the underlying type. Common use cases include dependency injection, creating mocks for unit testing, handling events, and defining callbacks.

Attack Interfaces in Go can be exploited to achieve polymorphic behavior when malicious code replaces a benign implementation at runtime. Listing 6.4 shows a scenario where an interface designed to call `'Execute'` on a safe type is subverted. By converting the instance of `'SafeType'` with `'UnsafeType'`(lines 15-16), the attacker ensures that the malicious implementation of `'Execute'` is invoked (line 17). The insidious nature of this attack lies in the ability to hide the malicious implementation behind an innocuous-looking interface. In addition, the subtle replacement can leverage dynamic or indirect assignment of types, potentially being concealed deep within the code, making it difficult to detect during code reviews or static analysis.

Listing 6.4. Indirect method invocation through interfaces.

```
1  type SafeInterface interface {
2      Execute()
3  }
4  type SafeType struct{}
5  func (b SafeType) Execute() {
6      // benign implementation
7  }
8  type UnsafeType struct{}
9  func (m UnsafeType) Execute() {
10     // malicious implementation
11 }
12 func main() {
13     var safeVar SafeInterface = SafeType{}
14     safeVar.Execute()       // Safe code exec
15     var unsafeVar SafeInterface = UnsafeType{}
```

```
16      safeVar = unsafeVar      // hidden conversion
17      safeVar.Execute()        // Unsafe code exec
```

E4. Unsafe Pointers

In Go, using safe abstractions such as arrays, slices, and maps ensures built-in checks that prevent common memory violations such as buffer overflows, null pointer dereferences, and out-of-bound memory accesses. However, to perform low-level programming tasks or to optimize certain performance-sensitive applications, the `unsafe` package can be utilized. This package allows for the definition of the type `'unsafe.Pointer'`, which bypasses Go's safety checks. For example, developers can perform pointer conversions, pointer arithmetic, and interpret the memory layout of complex data structures in ways that would otherwise be restricted.

Attack Safe usage of *unsafe pointers* is possible, however, several examples of patterns that introduce security risks exist [77, 179]. First, attackers can exploit unsafe pointers to perform unauthorized memory operations. As shown in Listing 6.5, unsafe pointers can be misused to create a function pointer variable and set its value to the address of an arbitrary function (lines 7-8), thereby effectively allowing the execution of any code at runtime.

Listing 6.5. Unsafe Pointers for Arbitrary Execution.

```
1  func targetFunction() {
2      // malicious code
3  }
4  func main() {
5      type FuncType func()
6      targetFunc := targetFunction
7      var funcPtr FuncType
8      funcPtr = *(*FuncType)(unsafe.Pointer(&targetFunc))
9      funcPtr()
10 }
```

Similarly, Listing 6.6 shows how unsafe pointers can be misused to access out-of-bound memory locations, potentially leading to information disclosure or memory corruption vulnerabilities.

Listing 6.6. Unsafe Pointers for Memory Access.

```
data := [4]int{1, 2, 3, 4}
ptr := unsafe.Pointer(uintptr(unsafe.Pointer(&data[0])) +
    unsafe.Sizeof(data[0])*4)
value := *(*int)(ptr)
fmt.Printf("Out-of-bounds value: %d\n", value)
```

E5. CGO Static Code Linking

CGO is a foreign function interface allowing for static linking of C code within Go packages. Developers can write C snippets in a preamble comment of a Go file, directly defining C functions or importing external C header files. Thus, by using a pseudo-package `'C'`, the defined C functions can be directly invoked within the Go program (shown in Listing 6.7). When the Go compiler parses an import of `'C'`, it invokes the C compiler on the defined C functions and links the C and Go object files. The CGO feature is commonly used to leverage existing C libraries to avoid error-prone re-implementations and bring performance benefits in Go [268].

Attack From a security standpoint, CGO introduces risks, mainly concerning memory safety. While Go has built-in protection [233] for common memory management errors, such as buffer overflows, dangling pointers, and memory leaks, C lacks these safeguards. Linking the C code in Go packages can reintroduce these vulnerabilities, which can be exploited by attackers. For example, memory violations can be used to gain unauthorized access or control over a system. In addition, attackers can manipulate function pointers in C to redirect execution to malicious code segments, bypassing Go's safety mechanisms.

Listing 6.7. C code invocation using CGO.

```
/* #include <stdio.h>
void cFunction() {
    // malicious code here
} */
import "C"
func main() {
    C.cFunction()
}
```

E6. Assembly Static Code Linking

Go supports static linking of assembly code [280]. Developers can write package-level assembly functions using Go Assembly. When compiling, any assembly file is assembled into object files by the Go assembler, and then linked into the Go objects. These functions can be used in Go files by defining a function stub with the same name as that specified in the assembly file. The assembly functions can then be called as any normal function in Go (shown in Listing 6.8). Assembly is commonly leveraged for code optimization or in security-critical domains such as cryptography [235].

Attack The complex syntax of assembly allows attackers to embed malicious code that can be challenging to audit due to its low-level nature. A lack of analysis support for assembly and the requirement of specialized knowledge (specifically in the go-specific assembly syntax) can leave this code poorly or completely unaudited. The use of empty functions as an assembly definition further conceals its underlying implementation. This makes assembly code a potent method for hiding malicious code.

Listing 6.8. Invocation of package-level defined assembly code.

```
1 func AsmFunction() int
2 func main() {
3     AsmFunction() // malicious function
4 }
```

E7. Dynamic Library Linking

In Go, developers can use the `plugin` package for dynamic loading and executing of external shared libraries within the main process of a running program. A plugin is a Go package with exported functions and variables built using the `'-buildmode=plugin'` option. The Go runtime resolves the function and variable symbols provided by the plugin dynamically, enabling direct function calls between the main program and plugin. The use of plugins eliminates the overhead of inter-process communication connected to e.g. file system operations.

Attack. While providing flexibility, plugins also introduce security risks. Plugins allow for changing the main program's behavior without recompiling it. An attacker can replace a plugin with a malicious version as shown in Listing 6.9. This allows an untrusted, potentially dangerous library to be loaded into the main process stealthily without the developer's knowledge.

Listing 6.9. Simplified example call to dynamic library function.

```
1 func main() {
2     pluginPath := "./malicious_plugin.so"
3     p := plugin.Open(pluginPath)
4     sym := p.Lookup("pFunc") // runtime resolution
5     fn := sym.((func()))
6     fn() // malicious function
7 }
```

E8. Dynamic External Execution

Dynamic external execution involves running an executable as a separate process. In Go. this can be achieved through the `os` and `syscall` packages, offering methods like '`exec.Command`' and '`exec.CommandContext`'. Additionally, lower-level functions such as '`syscall.ForkExec`', '`syscall.Exec`', and '`os.StartProcess`' provide more control over process execution. This feature is intended for directly execute an external binary or shell commands.

Attack Dynamic external execution poses security threats. For example, binaries executed through them may be opaque and limit human and static analyses, facilitating the hiding of malicious code. In addition, the possibility of invoking arbitrary commands that are potentially constructed dynamically can lead to OS command injection. This also introduces risks for denial-of-service attacks through fork bombs, i.e. recursive replication of processes.

6.2.4 Attack Vectors in the Malware Lifecycle

Based on the specific characteristics of each attack vector that we define, they can map to different stages of malware execution. During

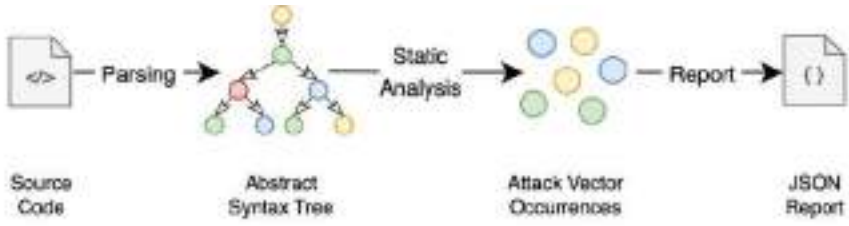


Figure 6.1. *GoSurf* Tool Architecture [58].

the *infection stage*, attack vectors categorized in the pre-build phase (Sec. 6.2.1), like using *generators* and *testing functions*, can be exploited to download preliminary malicious scripts or code, exfiltrate sensitive system information, or install backdoors. Subsequently, during the *persistence stage*, attack vectors exploitable at the initialization time (Sec. 6.2.2) like `init()` functions and global variables initialization can be used to establish a persistent presence on the infected system, for example, setting some registry entries or creating new scheduled tasks. As these functions are invoked automatically early in program execution, such malicious actions can occur stealthily without the developer's explicit invocation. Finally, all attack vectors we categorized as execution time (Sec. 6.2.3) can be exploited for the *payload execution stage* of malware, where the primary malicious activities are carried out.

6.3 GoSurf Design

In this chapter, we introduce *GoSurf*, a novel tool that analyzes the attack surface of Go modules according to the taxonomy of attack vectors described in Sec. 6.2. It reports the usage of language-specific features and programming idioms that can be leveraged for the corresponding attack vectors. *GoSurf* is meant to aid open-source dependency analysis and audit to find malicious code.

GoSurf follows a modular architecture consisting of three main parts: a parser, a static analysis engine, and a reporting component. The workflow is shown in Figure 6.1.

The *parser* component parses Go source code and constructs an Abstract Syntax Tree (AST). The *static analysis engine* uses the AST and a

set of rules to identify attack vectors based on the defined taxonomy. The *reporting component* consolidates the analysis results, presenting them in a user-friendly format. We implemented *GoSurf* in Go, using 793 lines of code, and leveraging Go standard libraries. The details of each component are explained in the following.

Parser The *GoSurf* parser component enables analysis of Go modules, potentially composed of several packages. First, the parser collects all packages for analysis, starting with a root module path specified as a local directory. It recursively traverses all subdirectories to identify the module packages, recognizing them by the presence of `.go` files containing valid package declarations (directive `package`). After gathering all packages from the root module path, the parser generates the abstract syntax tree (AST) representation of each Go file, using the Go standard libraries `go/token` and `go/parser`.

Static Analysis. *GoSurf* implements 12 analyzers to identify occurrences of every attack vector from our proposed taxonomy (Sec. 6.2). The analyzers use the AST representation to inspect its nodes, primarily using the `'ast.Inspect'` function from the `go/ast` package. The analyzers can be divided into three groups based on their targeted nodes for inspection: an invocation, declaration, or comment.

The first group of analyzers (*Constructor*, *Interfaces*, *Unsafe*, *CGO*, *Assembly*, *Plugin*, *Exec*) identify specific package and function invocation calls by inspecting *call expressions* (`'CallExpr'`). These analyzers search for function expressions belonging to an *identifier* (`'Ident'`) or *selector expression* (`'SelectorExpr'`). An identity is a standalone name, while a selector expression is a qualified identifier on the form `'X.Sel'`, where `'X'` is the package or receiver expression (e.g., a variable), and `'Sel'` is the identifier being selected (e.g., a function name). For example, the `Plugin` (Sec. 6.2.3) analyzer matches the AST node pattern of a call expression (`'CallExpr'`) with a selector expression having the package name `'plugin'`, selecting the method `'Open'` (`'plugin.Open()'`). Two invocation analyzers include a pre-processing step to identify the target method signature for inspection. The `Interfaces` (Sec. 6.2.3) analyzer collects all polymorphic methods by looking for functions with multiple

receiver types. Conversely, the `Assembly` (Sec. 6.2.3) analyzer collects all package-level assembly functions through regular expression matching in assembly files. Then, the invocation inspection technique is applied to these analyzers, identifying method calls to the previously collected method signatures.

The second group of analyzers (`GoTest`, `InitFunc`, `GlobalVar`, `Reflect`) analyzes declarations in the AST. The `GoTest` analyzer (Sec. 6.2.1) matches *function declaration* (`'FuncDecl'`) nodes that have names starting with `'Test'`, `'Fuzz'`, `'Benchmark'` or `'Example'`. Similarly, the `InitFunc` (Sec. 6.2.2) analyzer identifies function declarations named `'init'`. The `GlobalVar` (Sec. 6.2.2) and `Reflect` (Sec. 6.2.3) analyzers inspect *generic declaration* (`'GenDecl'`) nodes, looking for variable and constant declarations, and import declarations, respectively.

The third group inspects *comments* in the AST. This technique concerns the `GoGenerate` analyzer (Sec. 6.2.1), which identifies generators in `Comment` nodes with the prefix `'//go:generate'`.

Reporting. The reporting component of *GoSurf* records and reports the occurrences of an attack vector. When encountering a match in an analyzer, the location information is logged, including package name, file path, and line number. Additional context-specific information is included, depending on the attack vector, such as the name of a method invocation. All occurrences are aggregated into JSON format and counted before presenting the results in the CLI. Listing 6.10 shows a *GoSurf* report identifying usage of CGO and assembly.

Listing 6.10. Example JSON report.

```
[
 {
    "PackageName": "issue27340",
    "Type": "cgo",
    "FilePath": ".../issue27340/a.go",
    "LineNumber": 41,
    "MethodInvoked": "C.issue27340CFunc"
    },
    {
    "PackageName": "main",
    "Type": "assembly",
    "FilePath": ".../testdata/main.go",
```

```
13     "LineNumber": 8,
14     "MethodInvoked": "linefrompc"
15  }
16 ]
```

6.4 Usage of *GoSurf*

GoSurf is a tool meant to be integrated into supply chain security processes. We outline two primary use cases.

Pre-integration Auditing. Before integrating a third-party dependency, developers should conduct a thorough security audit. *GoSurf* offers a comprehensive overview of the risks associated with a dependency and highlights those riskier areas of the codebase with a higher number of attack vectors. It also provides a valuable security metric for dependency consumers, helping them prioritize those dependencies with smaller attack surface.

Version Update Monitoring The attack surface of a software package is dynamic, with new attack vectors potentially introduced with each new version release. *GoSurf* can be used to continuously monitor these changes, by comparing the attack surface of different versions of a package. This use case is particularly meaningful when a dependency with an initial trusted codebase is updated. Updates from untrusted authors with new attack vectors should be scrutinized. By using *GoSurf*, upstream dependencies can be effectively monitored, ensuring consumers stay informed about risks.

To sum up, *GoSurf* enhances the security posture in the Go ecosystem and identifies potential risks before actual software supply chain attacks.

6.5 Evaluation

In this section, we present initial experimental results obtained from analyzing *GoSurf* on real-world Go projects.

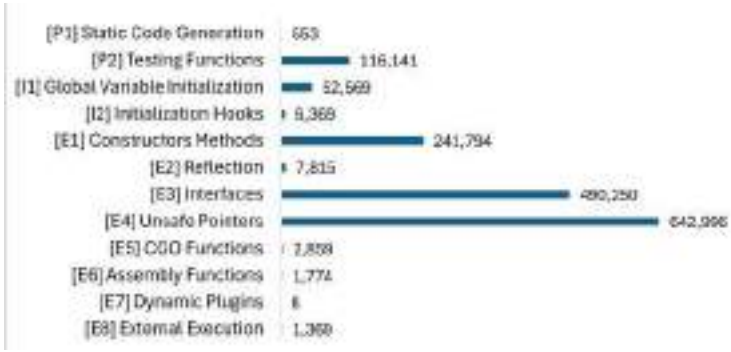


Figure 6.2. Go Attack Vector Prevalence in 500 Popular Go Projects. The rare ones are easier to forbid in order to secure a supply chain [58].

6.5.1 Attack Surface in Real-World Go Modules

In Sec. 6.2 we demonstrated the feasibility of the identified attack vectors, with executable proof of concepts. In this section, we conduct an experiment to evaluate the applicability of our proposed taxonomy, that is, that the 12 attack vectors can be successfully identified in real-world Go projects.

Methodology. To analyze the applicability of our taxonomy, we conducted a comprehensive analysis using *GoSurf* across 500 Go modules. We selected these modules based on their number of dependents, specifically the top 500 most imported packages, indicating their relevance and popularity. This selection criteria ensures that we focus on modules with widespread real-world usage, which are natural targets of supply chain attacks. The selected projects cover diverse categories such as cloud orchestration, networking, blockchain, monitoring, and CLI applications, sourced from `libraries.io`.

Results. Using *GoSurf*, we identified occurrences of some of the 12 attack vectors in each of the analyzed Go modules. Figure 6.2 reports the total number of occurrences for each vector across the 500 analyzed modules.

The results indicate that P2 (testing functions), E1 (constructors meth-

ods), E3 (interfaces) and E4 (unsafe) are the most frequently occurring attack vectors. Although their high prevalence may be connected to common benign Go programming practices, their usage call for additional auditing in a security-sensitive context. Overall, Table 6.2 shows that malicious actors have ample room to choose where to hide malicious code execution, as argued in Sec. 6.2.

Furthermore, the widespread usage of cross-language features such as C code and assembly (E5 and E6) (also reported in security-critical projects like Kubernetes and Go-Ethereum) suggests that developers may trade the built-in security mechanisms in Go for less secure code, growing the attack surface. Although less common, the use of plugins (E7) is also present. Here, the loading and execution of externally pre-built code requires thorough security analysis.

Now, let us consider Listing 6.11, demonstrating a real-world example of a dynamic external execution attack vector (E8) discovered by *GoSurf* in a Kubernetes dependency `mistifyio/go-zfs`. This small library implements a wrapper for ZFS command line tools and is maintained by 23 common GitHub users with no affiliation to the Kubernetes project.

It is important to clarify that the reported example is not a command injection vulnerability exploitable through a user-controlled input, but rather a potential attack vector, according to the definition in Section 6.2. It would be a good place for attackers to hide code in order to perform a supply chain attack. For example, a malicious actor would compromise a repository (e.g., a pull request that introduces a hidden payload or alters code in a subtle way), altering the dependency control flow to taint the arguments passed to `Run` (line 1), enabling execution of custom and lower-level OS commands. Here, the use of `exec.Command` presents a risk of a supply chain compromise, potentially affecting all the dependents of the `go-zfs` library, including Kubernetes.

Listing 6.11. Real-world attack vector.

```
1 func (c *command) Run(arg ...string) ([][]string, error) {
2     cmd := exec.Command(c.Command, arg...)
3         ...
4 }
```

For attackers, it is tempting to target widely-used dependencies to maximize the impact on downstream applications. In addition, depend-

encies with a larger attack surface are more likely to be targeted because they provide more opportunities for hiding the execution of malicious code. Therefore, these dependencies require closer scrutiny.

To sum up, *GoSurf*'s results validated the applicability of our taxonomy. In Sec. 6.2, we demonstrated the feasibility of each attack vector by developing PoCs for them. In this section, we confirm the increased security risks associated with all 12 vectors by showing their prevalence in real-world Go projects. This effectively motivates the need for additional scrutiny and prioritization of these vectors during security audits.

6.5.2 Attack Surface over Versions

The attack surface of a software package is dynamic, with new attack vectors potentially introduced with each new version release. *GoSurf* can be leveraged to continuously monitor these changes by analyzing and comparing multiple versions of a package. In this experiment, we aim to demonstrate the dynamic nature of the attack surface and highlight the importance of regularly auditing dependencies for emerging threats, reasoning the *GoSurf*'s use case described in 6.4.

Methodology. To examine how the attack surface evolves in an actively developed project and understand the impact of software updates on supply chain security, we used *GoSurf* for differential analysis. We evaluated the last five major releases of the popular Kubernetes project (versions 1.26 through 1.30) to investigate the prevalence of different attack vectors as the codebase evolved. The selected releases cover a time window of around 16 months.

Results. The results obtained from running *GoSurf* on five different versions of Kubernetes are shown in Table 6.1. The reported occurrences suggest that the attack surface is not static and can significantly vary as new features are added, and code is refactored over release cycles.

For example, the introduction of *structured authorization configuration* likely contributed to the increase in interfaces (E3) from Kubernetes v1.28 to v1.29 [163], as this feature allows for an extensible and decoupled authorization mechanism. Similarly, the *in-place update of pod resources*, introduced in the same upgrade, potentially contributed to the increased

Table 6.1. Vector usage in 5 different releases of Kubernetes

	v1.26	v1.27	v1.28	v1.29	v1.30
P1	106	95	104	147	119
P2	10,067	10,291	10,501	10,770	10,339
I1	35	35	36	37	36
I2	8319	8,304	8,320	8,336	1,108
E1	43,701	43,102	43,822	44,389	38813
E2	1,543	1,526	1,540	1,568	1,528
E3	93,076	87,700	89,849	90,901	85277
E4	7,769	7,744	7,883	7,865	8,105
E5	820	792	795	797	803
E6	1,495	1,494	1,494	1,495	1,495
E7	1	1	1	1	1
E8	242	236	236	244	230
LOC	3,967,186	3,855,352	3,934,131	4,004,993	3,728,835

usage of reflection (E2), as it might require dynamically accessing pod structures.

On the other hand, it is reasonable to observe a decrease in the attack vectors between Kubernetes versions 1.29 and 1.30, as the codebase has reduced in size. This reduction can often be attributed to the stabilization of features and the removal of redundant code, suggesting that refactoring can significantly reduce the attack surface for Go packages.

In conclusion, *GoSurf* is a tool for auditors to continuously monitor the attack surface of dependencies. As software dependencies are updated over time, new attack vectors may be introduced, potentially increasing the risk for downstream consumers. Continuously monitoring these changes is crucial for maintaining a secure software supply chain.

6.6 Limitations

While *GoSurf* proved effective in mapping the presence of attack vectors across real-world Go projects, its design deliberately favors broad coverage over precision and thus comes with important limitations.

Over-approximation and False Positives. *GoSurf* operates as a purely static, syntactic analysis that flags language constructs and idioms associated with our taxonomy of attack vectors. By design, these findings represent *potential* execution vectors rather than confirmed vulnerabilities. The tool does not perform data-flow, control-flow, or taint analysis, nor does it reason about input controllability, reachability, or context-specific semantics. As a result, *GoSurf* can report a high number of false positives: many flagged locations are benign uses of powerful features that *could* be abused, but are not currently exploitable. Consequently, *GoSurf* is not suited for direct vulnerability detection or malware classification; instead, it should be interpreted as an attack-surface profiler that highlights code regions deserving closer scrutiny.

Opportunities for Deeper Analyses. The coarse-grained nature of *GoSurf*'s findings also outlines clear directions for future work. One avenue is to treat *GoSurf*'s reports as an entry point for more precise, layered analyses on the flagged critical paths. For example, capability-oriented tools such as Capslock [124] could be integrated to perform fine-grained analysis of how flagged execution vectors interact with system capabilities, differentiating harmless uses from those that cross sensitive privilege boundaries. Similarly, differential analysis across versions, building on *GoSurf*'s version-comparison mode, could focus deeper analyses (e.g., symbolic execution, taint analysis, or targeted fuzzing) on newly introduced or significantly changed attack vectors. In this way, future work can use *GoSurf* not as a standalone vulnerability detector, but as a front-end for prioritizing and steering more expensive security analyses towards the most security-relevant parts of a Go codebase.

6.7 Discussion

Software supply chain attacks targeting open-source dependencies represent one of the most pervasive and difficult-to-contain security challenges in modern software ecosystems. This chapter addressed the problem from the perspective of attack surface reduction. To characterize this domain, we introduced the first Go-specific taxonomy of supply chain attack vectors, systematically identifying twelve concrete execution vectors distrib-

uted across the pre-build, initialization, and runtime phases. This taxonomy captures how Go's language constructs, package manager features, and build mechanisms can be repurposed by adversaries to inject, trigger, or conceal malicious behavior. Building upon this foundation, we developed *GoSurf*, a static analysis tool that makes the taxonomy actionable. Our evaluation on real-world Go projects demonstrated both the applicability of the taxonomy and the effectiveness of *GoSurf* in surfacing previously overlooked risks. In doing so, *GoSurf* extends the state of the art beyond vulnerability scanning and rule-based linting, offering a structured method to quantify and reduce the software supply chain attack surface before deployment.

Taken together, these contributions address the static dimension of supply chain hardening: understanding and identifying the latent execution capabilities that can later be exploited. However, as discussed in Section 2.1.4, many threats in Go manifest dynamically, through conditional logic, obfuscated payloads, or runtime reflection—beyond the reach of static inspection. This motivates the subsequent chapters of this dissertation, which extend the defense boundary from pre-deployment detection to runtime enforcement. The next stage of this research explores how behavioral tracing and capability enforcement mechanisms, such as *GoLeash*, can further reduce the attack surface by constraining package behavior during execution, bridging the gap between static assurance and dynamic resilience.

Chapter 7

Mitigating Software Supply Chain Attacks at Runtime

7.1 Overview

As software supply chain attacks continue to target development ecosystems, static analysis alone is insufficient to ensure runtime integrity. Once a dependency has been compromised, malicious code can still execute within trusted applications, exfiltrating data or performing unauthorized actions. The Go ecosystem, used in critical infrastructures such as Kubernetes, lacks mechanisms to enforce least privilege at the package level during execution, leaving a wide gap between static assurance and runtime enforcement.

This chapter presents *GoLeash*[1], a framework and research prototype for detecting and mitigating malicious behavior in Go applications at runtime. Grounded in the principle of least privilege [249, 262], GoLeash leverages the eBPF framework [94] to monitor system call activity and automatically infer which Go packages access which capabilities. In analysis mode, it profiles legitimate executions to generate a minimal set of required capabilities for each package; in enforcement mode, it blocks any package attempting to access capabilities beyond that baseline. For instance, a data-processing package attempting to initiate a network connection would be immediately flagged and contained. Unlike existing

[1] Available as open-source at [41]

sandboxing or syscall filtering mechanisms (e.g., seccomp [285], gVisor [130]), which operate at the process or container level, GoLeash performs enforcement at the package level, achieving much finer granularity. This distinction enables it to isolate malicious dependencies without penalizing the rest of the application.

We evaluate GoLeash by injecting stealthy malicious behaviors into complex Go applications, including Kubernetes. GoLeash successfully identifies compromised packages in 98% of cases, compared to 32% detection accuracy using coarse-grained monitoring, and remains effective even under code obfuscation. The system introduces minimal runtime cost, with an average system call latency of 3.17 ms and an overall performance overhead of 9.3%.

7.2 Threat Model

GoLeash is designed to mitigate software supply chain attacks targeting third-party Go dependencies. Our threat model assumes attackers compromise initially trusted dependencies through malicious updates, typically via compromised developer credentials or repository hijacking [261]. The primary attacker objective is to inject malicious code into legitimate-looking packages to perform unauthorized operations at runtime, such as accessing and manipulating the filesystem, establishing network communications, and executing arbitrary commands. That is, third-party dependencies are considered untrusted and potentially compromised, as they can embed arbitrary code execution capabilities.

Attackers can add malicious capabilities in pure Go code, either using obfuscation or not. Attackers in our model may also employ sophisticated techniques beyond Go code [58], using mechanisms such as CGO integration (C binding), inline assembly, dynamically loaded plugins, and execution of external binaries. Such advanced methods evade detection from static analysis tools, as malicious behaviors only become observable during actual program execution.

The Go run-time environment is assumed to be trusted and uncompromised. This implies that a threat actor cannot manipulate the context of system calls or compromise the integrity of the runtime's stack traces, and that control flow proceeds as intended (i.e., no external interference or

tampering affects how Go dispatches function calls, manages stack traces, or schedules goroutines).

We do not cover attacks executed during earlier phases of software development, such as attacks triggered by test functions, Go generate scripts, Makefiles, or other build and deployment tooling. These attacks primarily compromise developer machines rather than production systems.

> Our threat model is unauthorized runtime behavior in production, introduced via third-party Go dependencies compromised with malicious code.

7.3 GoLeash Design

In this section, we present the design requirements, key design choices, and implementation details of GoLeash. GoLeash is designed to meet the following requirements, which are essential for the practical adoption of the approach:

- Infer package-level security policies without developer manual work.
- Enforce security policies without changes to source code, build pipelines, and the Go runtime.
- Handle real-world, complex applications, including ones involving multiple processes and binaries.
- Detect malicious dependencies even when the malicious behavior is obfuscated.

Figure 7.1 illustrates the high-level architecture of GoLeash. The system consists of a kernel-space tracing component and a user-space analysis engine. The workflow includes the following stages. (1) GoLeash inserts tracing probes in the kernel to capture system call (syscall) events. (2) These events are pushed into a ring buffer, and asynchronously collected in user space, where (3) the stack trace is resolved. (4) GoLeash maps the stack trace to the originating Go packages, and (5) classifies the syscall into a capability. Finally, (6) GoLeash either uses the event to build a policy (*analysis mode*), or it checks whether the event matches against

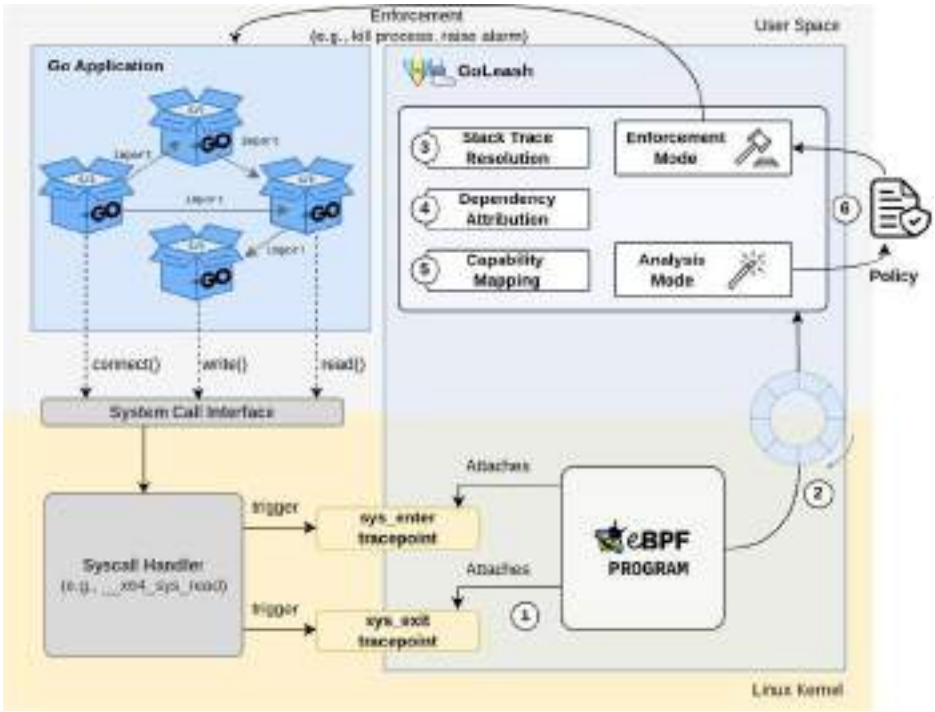


Figure 7.1. GoLeash Architecture [61].

a policy to detect violations (*enforcement mode*). In the first case, the allowlist is saved for later use. In the second case, enforcement actions are triggered. This architecture allows GoLeash to enforce least-privilege boundaries across dependencies, as opposed to coarse-grain process-based privileges. The system is designed to operate with no changes to build and deployment pipelines, it does not require recompilation or dedicated instrumentation of the protected application.

7.3.1 System Call Tracing

At the core of GoLeash lies a low-level monitoring infrastructure built on top of the Extended Berkeley Packet Filter (eBPF) [94] framework. eBPF is a Linux kernel technology that executes small, sandboxed programs in kernel space. eBPF offers a versatile framework for dynamic

tracing and performance analysis. By attaching eBPF programs to kernel hooks, one can analyze kernel events at runtime with minimal overhead. This includes capturing information on system call IDs and arguments, and the current stack trace. eBPF is an excellent tool for security. It has been used to implement fine-grained policies and analyses on system-wide and application-specific behavior [104, 69, 272, 144]. Since eBPF is embedded in the kernel, it ensures that telemetry data collection remains efficient and isolated.

GoLeash attaches eBPF programs to syscall-related tracepoints, allowing it to intercept every system call made by the target application in real time. A Linux tracepoint provides a static hook point to call a function provided at runtime. Specifically, eBPF programs are attached to Linux kernel tracepoints to capture syscall invocations [284].

Each captured syscall event includes two key pieces of information: the *syscall identifier* and a *stack trace identifier*, which references the user-space call stack that led to the syscall.

To reduce noise and ensure that only relevant syscalls are captured, GoLeash filters events based on the *command name* of the traced binary, and tracks all associated PIDs. This design supports real-world scenarios, such as multi-process Go applications, and horizontally-scaled replicas in containerized environments.

7.3.2 Dependency Attribution

After system call events are captured in kernel space, GoLeash's user-space component analyzes these events, to attribute each syscall to the specific Go package responsible for triggering it. This attribution is critical for building precise capability profiles and precise policy enforcement at the package level. Each syscall event delivered from the kernel includes a stack trace identifier, which corresponds to a snapshot of the user-space call stack captured at the time of invocation. In user space, GoLeash resolves this identifier into a full stack trace by referencing the `stack_id` and its associated memory snapshots. To interpret the raw return addresses in the stack trace, GoLeash maps each address to the corresponding function name and Go package import path. This symbol resolution process begins by parsing the target binary's ELF symbol table, which provides a mapping from memory addresses to symbol names in unstripped binaries.

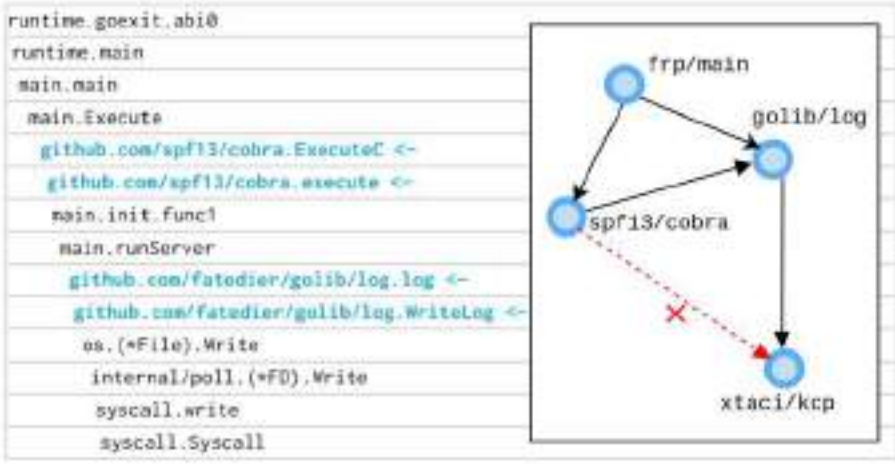


Figure 7.2. On the left: example stack trace with highlighted frames that are part of the call path responsible for the write syscall. On the right: dependency graph of the corresponding application being traced [61].

Once the stack trace is resolved, GoLeash traverses it from the most recent frame backwards to identify the Go package from which the syscall originates. In Go programs, system calls often pass through multiple layers of abstraction, including standard library functions (e.g., from `net/http`, `os`, and `io` [122]) and runtime helpers, before reaching the kernel. These intermediate layers are part of Go's trusted infrastructure and are not considered responsible for the syscall. GoLeash should not misattribute behavior to trusted components. Hence, GoLeash does not assign the syscall to the topmost stack frame. Instead, it scans the full trace to find the first frame originating from an application-defined or third-party package function, and considers that frame as the initiator of the syscall. For instance, in the trace shown in Figure 7.2 (on the left), where `os.Write` and other standard packages wrap the syscall, GoLeash skips them and correctly attributes the syscall to the frame of third-party package `fatedier/frp/server`. In addition to identifying the responsible frame, GoLeash also saves the full chain of Go packages that led to the syscall, for call path analysis discussed later in subsection 7.3.5.

7.3.3 Capability Mapping

GoLeash defines a capability taxonomy (Table 7.1) that maps low-level syscalls to semantically meaningful categories of system functionality. The taxonomy aggregates syscalls that operates on the same resource and that serve the same function, and yet differentiates between operations that have different security implications. For instance, filesystem-related syscalls are split into read and write capabilities, allowing developers to grant read access without permitting file modifications. Likewise, networking is broken down into socket creation, connection establishment, and packet transmission. This granularity enables expressive yet narrowly scoped policies, significantly reducing the risk of unintended or malicious behavior. The syscall-to-capability mapping is manually curated and verified against official Linux syscall documentation [286] to ensure accuracy. In our implementation, we cover all syscalls in Linux kernel version v6.7. The detailed mapping is provided within our implementation of GoLeash [41]. During both analysis and enforcement, GoLeash uses this mapping to translate raw syscall activity into high-level capability usage.

7.3.4 Analysis Mode

Once system calls are attributed to specific capabilities and the invoking package is identified through stack trace resolution, GoLeash aggregates this information to build a capabilities *allowlist* for each package. These allowlists are constructed in the *analysis mode*, where the application is executed in a trusted environment, using integration tests or representative workloads to exercise the intended behavior of the application. This allowlist acts as a behavioral contract: any future execution is expected to conform to these observed patterns. Once reviewed and approved, the allowlist becomes the *policy* used for enforcement.

For each traced package exercised during the execution, the allowlist records the set of capabilities it invoked, as well as all observed call paths that led to each capability. These call paths capture the broader execution context: they consist of the ordered sequence of Go packages extracted from the stack trace, with the traced package acting as the terminal caller responsible for the capability.

Table 7.1. Capability Taxonomy Used in GoLeash

Category	GoLeash Caps	Description	Example Syscalls
File Capabilities	`CAP_FILE`	Manage file descriptors	`close`, `poll`
	`CAP_READ_FILE`	Read data from files	`read`, `open`, `stat`
	`CAP_WRITE_FILE`	Write or append data to files	`write`, `writev`
	`CAP_CREATE_FILE`	Create new files or directories	`mkdir`, `creat`
	`CAP_DELETE_FILE`	Remove existing files or directories	`unlink`
	`CAP_FILE_METADATA`	Modify file metadata (permissions or ownership)	`chmod`, `chown`
Network Capabilities	`CAP_CONNECT_REMOTE`	Initiate outbound connections to remote endpoints	`socket`, `connect`
	`CAP_LISTEN_LOCAL`	Bind to local ports to accept incoming connections	`bind`, `listen`
	`CAP_SEND_DATA`	Transmit data over established network connections	`sendto`, `sendmsg`
	`CAP_RECEIVE_DATA`	Receive data from network connections	`recvfrom`, `recvmsg`
Execution Capabilities	`CAP_EXEC`	Launch new processes or threads	`clone`, `execve`
	`CAP_TERMINATE_PROCESS`	Terminate running processes or threads	`exit`, `kill`
System State and Configuration	`CAP_READ_SYSTEM_STATE`	Access system configuration or status information	`getpid`, `getitimer`
	`CAP_WRITE_SYSTEM_STATE`	Modify environment variables or system settings	`setuid`, `setgid`
	`CAP_RESOURCE_LIMITS`	Adjust process or system resource limits	`setrlimit`
Memory Operations	`CAP_MEMORY_MANIPULATION`	Alter memory regions or mappings at runtime	`munmap`, `mmap`
	`CAP_DIRECT_IO`	Perform low-level I/O operations on devices or memory	`ioctl`

Formally, the allowlist $\mathscr{A}$ is defined as:

$$\mathscr{A} = \{(P_i, \{(C_{ij}, \mathscr{T}_{ij})\})\}$$

- P_i is a traced package,
- C_{ij} is a capability invoked by P_i,
- $\mathscr{T}_{ij} = \{T_{ijk}\}$ is the set of call paths observed for capability C_{ij},
- each call path T_{ijk} is an ordered list of Go packages:

$$T_{ijk} = [Q_1, Q_2, \ldots, Q_n, P_i]$$

with Q_k denoting intermediary packages and P_i as the final traced package.

To optimize for performance and memory efficiency, GoLeash stores each observed call path as a hash of the array $\mathscr{T}_{ij}$. This compact representation allows for fast lookups at run-time while preserving the ability to verify complete calling contexts. Recalling the example in Figure 7.2, during analysis, GoLeash saves the entire sequence of legitimate call paths executed, such as `[cobra` $\rightarrow$ `golib` $\rightarrow$ `write]`.

Policy Management in the Development Process The construction of allowlists is designed to be iterative and non-disruptive. As new functionality is added to the application, GoLeash can be re-run to observe and record additional capability and call paths, and incorporate them into the existing policy. This makes it possible to start with a conservative baseline and refine the policy over time without requiring a complete regeneration. Additionally, developers can manually audit the allowlist and append entries when necessary, such as when preparing for production deployment after a review of expected system behavior. For example, developers can review which Go packages run external executables or establish outbound network connections, and freeze the allowlist after the review, so that no other package will be able in the future to stealthly introduce such operations for malicious purposes.

In typical development workflows, developers would often modify allowlists for major version updates of a dependency. Minor version updates usually involve bug fixes or refactoring, and by default do not call for updating policies. To sum up, GoLeash policies should be updated and re-audited whenever a dependency publishes a major version, or when new privileged operations are explicitly added in Changelogs.

A discussion of how human decision-making and policy review practices may affect the effectiveness of GoLeash in real deployments, and how this constitutes a threat to validity of our empirical results, is provided in Section 7.6.

7.3.5 Enforcement Mode

Once a policy has been constructed and approved during development, GoLeash can be used in the enforcement mode to protect the running application from unauthorized behaviors. The same eBPF-based tracing infrastructure is reused to intercept system calls; however, instead of gathering these observations for analysis, GoLeash now checks each syscall in real-time against the policy. Enforcement is made at runtime by validating both the origin of the syscall and the capability being exercised.

For every intercepted syscall, GoLeash resolves its associated capability and identifies the terminal package responsible, using the same attribution and classification logic from analysis. The syscall is then validated against the policy: if the `(package, capability)` pair is present, and the corres-

ponding call path matches one of the approved sequences stored for that capability, the syscall is allowed to proceed uninterrupted. Otherwise, GoLeash flags the event as a policy violation and triggers a configurable enforcement action.

By validating not only the invoking package but also the full call path in the stack trace, GoLeash supports context-aware enforcement. This defends against confused deputy attacks [197], when a restricted dependency may attempt to trigger a privileged capability indirectly by invoking a more permissive package within the same application. In such cases, even if the terminal package is authorized for a capability, the unapproved calling context causes the request to be denied. This ensures that both the origin and the execution context of every syscall align with the previously observed and trusted policy. In the example of Figure 7.2, an attacker could compromise `spf13/cobra` to import a function from `xtaci/kcp`, triggering a `SEND_DATA`, which was not part of `cobra`'s original capability set (see also Appendix A). GoLeash prevents the invocation path between cobra and `kcp`, since it is not part of the policy.

GoLeash supports multiple enforcement strategies, allowing it to be used for forensics and response purposes. In forensics use cases, violations are logged for postmortem analysis, enabling developers to assess suspicious behaviors without disrupting application functionality. In response use cases, GoLeash can terminate the offending process entirely, preventing potential exploitation in real time. These strategies can be configured per environment, supporting progressive hardening.

7.3.6 Advanced Support

Support for Trusted Go Internals. It is important to consider that the Go runtime itself can initiate system calls, independently from application logic, such as, for scheduling, I/O polling, and garbage collection. GoLeash distinguishes between system calls originating from the Go runtime from those issued by packages of the application and modules. These runtime-initiated events are excluded from enforcement, by examining the stack trace associated with each system call. If the entire call stack contains only frames from Go's runtime and does not include any package of the application or modules, the system call is treated as coming from trusted infrastructure and excluded from analysis. This design ensures

that only behavior explicitly caused by the application or its dependencies is subject to capability enforcement, reducing the overhead and avoiding false positives.

Support for Evasion Techniques. The syscall attribution mechanism implemented in GoLeash is robust even in the presence of advanced attack vectors that use defense evasion techniques, including obfuscation. Techniques such as encoded Go code, CGO bindings, inline assembly, and dynamically loaded plugins may attempt to execute malicious operations. However, all such operations ultimately result in syscalls that pass through the kernel. Since the stack includes the originating Go package, GoLeash can identify the dependency involved in the syscall. Moreover, GoLeash is able to handle malicious code that hides its behavior by executing existing binaries on the target system (*living-off-the-land* attacks), as in MITRE ATT&CK *T1036 (Masquerading)*, as well as through obfuscation and control flow manipulation strategies commonly seen in *T1574 (Hijack Execution Flow)*, as described later.

Support for Exec-based Control Transfer. Go applications commonly invoke external binaries to delegate tasks, such as running system utilities, interacting with non-Go components, and launching helper tools. This pattern is supported directly by the Go standard library via wrappers around `exec()`. However, this behavior introduces challenges for security monitoring. The `exec()` syscall replaces the current process image with a new binary while retaining the same PID, effectively shifting execution to code that may not be vetted or trusted. Attackers can exploit this to bypass runtime controls from within a compromised package. This tactic aligns with MITRE ATT&CK techniques *T1055 (Process Injection)* and *T1543 (Create or Modify System Process)*. To handle this safely, GoLeash analyzes both `sys_enter` and `sys_exit` events for the same `exec()` syscall. A pending "external binary execution" event is recorded at the syscall entry, and only committed if the syscall completes successfully. This avoids recording syscalls from failed or aborted transitions. It must be noted that the external binary is not necessarily a Go program. Thus, after a successful exec, GoLeash tracks syscalls using a flat (i.e., dependency-unaware) allowlist for the process as a whole, indexed by

the new executable's name. During this transition, residual syscalls, such as those issued by the Go runtime or by parallel threads active at the moment of the `exec()`, are filtered out.

Support for Multi-process and Multi-Binaries. GoLeash handles runtime behaviors that complicate syscall attribution in Go, such as process forking and multi-binary applications. Supporting these behaviors is essential, as they are common in modern Go systems. Forking is used to spawn workers, isolate tasks, and handle concurrent requests, while multi-binary applications assign distinct responsibilities to separate executables. This design is especially prevalent in distributed systems. For example, Kubernetes runs several binaries within a single node, and often deploys multiple replicas of the same binary for redundancy and load balancing. These behaviors are also security-relevant. Attackers may fork subprocesses to evade runtime monitoring or enforce stealth, aligning with MITRE ATT&CK techniques such as *T1059 (Command and Scripting Interpreter)*. GoLeash counters this by dynamically tracking all processes spawned across all binaries in the target application, including those created via `fork()`, ensuring that syscall attribution and policy enforcement remain consistent across the entire process tree.

7.4 Implementation

We implemented GoLeash using the C and Go languages. The codebase includes 1,190 lines of code, split between the eBPF tracing component (kernel-space) and the Go-based analysis and enforcement engine (user-space).

The kernel component uses the `cilium/bpf2go` library [68] to attach probes that capture syscall events. Each event includes a hash-based ID derived from the user-space stack trace, enabling efficient, deduplicated lookup for execution paths. Events are relayed to user space via a ring buffer to ensure low-overhead, non-blocking communication. The user-space component resolves stack traces, maps instruction addresses to Go symbols and Go package paths, and enforces capability policies. This process leverages symbol information embedded in Go binaries, which includes fully qualified package paths, allowing for accurate attribution without

source code access. GoLeash targets x86-64 Linux systems and requires a kernel version 5.7 or later for full eBPF tracing support.

7.5 Experimental Analysis

We conduct a systematic, large-scale experimental evaluation of GoLeash, structured around the following research questions.

RQ1: How effectively does GoLeash detect malicious code in third-party dependencies?
We investigate GoLeash's ability to identify unauthorized use of system capabilities as part of software supply chain attacks. The experimental protocol consists in simulating realistic supply chain attacks, by injecting malicious behaviors into benign Go projects, and measuring GoLeash's detection rate.

RQ2: How robust is GoLeash in detecting obfuscated malicious behavior? Typically, malicious software is hidden through obfuscation techniques, to make it more difficult to detect by malware detectors. Thus, we assess whether obfuscation techniques can evade GoLeash's enforcement mechanism.

RQ3: What is the performance overhead introduced by GoLeash?
We measure the overhead introduced by GoLeash in terms of system call latency and overall execution time of traced applications.

RQ4: How does GoLeash compare with static capability analysis? GoLeash leverages dynamic analysis for capability attribution, complementing static approaches. Therefore, we quantify the overlap and differences between GoLeash and a state-of-the-art static capability analysis tool.

Experimental Targets. We evaluate GoLeash on real-world, complex Go software projects. We select projects that meet the following criteria: 1) they have more than 10000 stars, 2) they are actively maintained, 3) they come a test suite or a workload to exercise their core functionalities, 4) they are security sensitive Applying this strict criteria result in

the following five projects: *kubernetes* (k8s) [168] for container orchestration, *etcd* [100] for key-value storage), *coredns* [75] and *frp* (fast reverse proxy) [106] for networking, and *go-ethereum* (geth) [101] for blockchain infrastructure.

Those applications are complex and some contain multiple binaries. We include all binaries in our evaluation. For example, in the case of Kubernetes, we analyzed the five main control plane binaries:*kube-apiserver*, *kube-controller-manager*, *kube-scheduler*, *kube-proxy*, *kubelet*.

Malicious Code Injection. Malicious behaviors in supply chain attacks typically abuse system-level capabilities, such as establishing outbound network connections to exfiltrate data, accessing the file system to steal information, and executing system commands to further compromise the system. To the best of our knowledge, there is no existing dataset of malicious Go packages. We tried to access the few malicious ones reported in blog posts, by did not succeeding in obtaining the code.For our evaluation, we tackle this lack of data by building synthetic malicious packages according to the following sound methodology.

Malicious packages are constructed by selecting a package from the dependency graph of the application under study, and by injecting a piece of malicious code into this target package. The selection of packages to consider is uniformly random, ensuring the absence of bias.

First, we surveyed malicious code patterns from established datasets in npm and PyPi [83, 218]. Next, we adapt those patterns to the Go language through five different malicious behavior injectors, spanning data exfiltration, remote file infiltration, information stealing, code injection and execution. Our malware implementations adopt the same Go packages also adopted by known campaigns against Go projects, as reported by technical blogs [53, 54, 244]. Table 7.2 summarizes these behaviors, and maps them to the relevant entries of the MITRE ATT&CK knowledge base.

We automate the injection of malicious code through rewriting the source code of the target package. We developed a tool to analyze the abstract syntax tree of the target package, and to weave malicious code snippets within an existing function of the package, without disrupting the existing functionality and structure of the target package. During experi-

Table 7.2. Injected malicious behaviors.

ID	Behavior	MITRE ATT&CK
M1	Base64-encoded command execution for reverse shell access	T1059, T1140
M2	Exfiltrate and transmit system configuration data to a remote server	T1552, T1082, T1041
M3	Information stealing from user applications	T1115
M4	Inject a shared library into current process	T1055.001
M5	Infiltrate malicious files	T1105

ments, we injected each malware type into every exercised package (targeting a randomly selected exercised functions). Our malicious code injector is made open source for future research. All malicious packages generated during experiments are publicly available in our replication package [41].

System Configuration. All experiments are conducted on a machine with an Intel Core i7-1260P CPU, 16GB of RAM, running Ubuntu 22.04, with Linux kernel version 6.7, and Go version 1.24.1.

7.5.1 Effectiveness Against Malicious Behavior

Objective. The goal of this experiment is to evaluate GoLeash's effectiveness in mitigating supply chain attacks by detecting unauthorized system capability usage introduced through compromised third-party packages. We aim to show that GoLeash's per-package enforcement can accurately flag malicious behaviors, even when injected into deep transitive dependencies. As a baseline, we compare against the state-of-the-art process-level enforcement model, such as Seccomp [285], which handles the entire application as a monolith. A detection is considered successful when GoLeash successfully detects that a malicious package exercises a malicious capability absent from the policy.

Methodology. We first use GoLeash in *analysis mode* to generate per-package policies. This is done by executing each target application with its end-to-end workload while GoLeash traces all invoked packages. These workloads are designed to reflect the application's intended, benign behavior. To compare GoLeash's per-package enforcement with traditional

sandboxing models, we implement a *process-level enforcement baseline*. This baseline aggregates all observed capabilities into a flat, application-wide policy, without attributing them to individual packages.

After policy generation, we construct a dataset of malicious variants of the target applications. For each variant, we rebuild the application with exactly one malicious code snippet at a time, injected into a single target package. The injection point is randomly selected from functions that are actually executed, which are profiled using our tracing infrastructure. It is worth noting that not all packages in the dependency graph are actually reachable by the workload. For example, the `net` Go package may import `crypto` for encrypted communication, but the application using `net` might not use encryption features. Therefore, we limit injections to packages that are actually active during execution. This ensures that the malicious code is reachable and that it represents a realistic attack. We leverage the injection approach described in Section 7.5. In total, we produced *3,265* unique malicious application variants.

The effectiveness of GoLeash is measured using the *detection rate*, defined as the percentage of malicious packages flagged by GoLeash out of the total number of created malicious packages for a given application. For comparison, the same metric is applied to the process-level baseline. Under this coarse-grained model, a variant is only detected if the injected code uses capabilities not observed anywhere in the application during baseline execution. Malware IDs used in the evaluation are defined in Table 7.2, and the *injections* column indicates the number of application packages individually injected with each malware for testing.

Results. Table 7.3 summarizes the results of this experiment. As previously discussed, we profile which packages are actually executed (e.g., 441 in the case of k8s), and inject five types of malicious code into each target package. In total, we obtain $3,265$ malware injections.

GoLeash successfully detects malicious behavior with high accuracy. The average detection rate across all projects is 98%. This is much higher than the 32% detection rate achieved by the application-wide baseline approach. GoLeash achieves a perfect 100% detection rate on two different malware categories: M1 and M3. Importantly, the detection rate never dropped below 87%. GoLeash effectively identifies deviations from normal

Table 7.3. Detection Rate of GoLeash vs Baseline. Malware IDs are defined in Table 7.2. The *"Injections"* column indicates the number of malicious application variants.

Project	Malware	Injections (#)	Baseline (%)	GoLeash (%)
k8s [168]	M1	441	100	100
	M2	441	0	99.77
	M3	441	100	100
	M4	441	0	99.32
	M5	441	0	99.55
etcd [100]	M1	62	100	100
	M2	62	0	98.39
	M3	62	0	100
	M4	62	0	91.94
	M5	62	0	98.39
coredns [75]	M1	20	100	100
	M2	20	0	100
	M3	20	100	100
	M4	20	0	95
	M5	20	0	100
frp [106]	M1	47	100	100
	M2	47	0	97.87
	M3	47	100	100
	M4	47	0	87.23
	M5	47	0	95.74
geth [101]	M1	83	100	100
	M2	83	0	97.59
	M3	83	0	100
	M4	83	0	91.57
	M5	83	0	97.59
Aggregate		3,265	32%	98%

execution, by detecting when the injected malicious code uses capabilities that are not part of the policy generated with benign executions. Our results indicate that certain sensitive capabilities, such as initiating outbound network connections or spawning external commands, used by malware, are particularly effective indicators.

In contrast, the coarse-grained, process-level baseline model was significantly less effective. By aggregating all capabilities into a single policy, process-level enforcement often allows stealthy malware to go undetected. This happens when the malicious code uses capabilities that are also legitimately used by the target package. The baseline is able to detect malware

M1 and M3, which leverages the rarely-used capability `RESOURCE_LIMITS` capability, which is absent in the original applications.

Finally, we observe that GoLeash enforcement incurs *zero false positives* in our experiments. However, it is important to note that, in general, dynamic analysis can be affected by false positives if the security policy is trained with a workload that is not representative of the operational stage. We further discuss this aspect in Section 7.6.

> We have performed a large-scale experiment with 3265 malicious packages injected in five real-world Go projects. GoLeash stops 98% of the simulated software supply chain attacks. GoLeash's per-package enforcement offers precise and robust defense against software supply chain attacks.

7.5.2 Effectiveness Against Obfuscated Attacks

Objective. Malicious actors often attempt to hide dangerous logic via obfuscation techniques, making it harder for security tools to detect unauthorized actions [298, 220]. This experiment evaluates whether obfuscation techniques impact the ability of GoLeash to perform per-package capability enforcement.

Methodology. As in RQ1, we begin by running each target application with its normal workload in GoLeash analysis mode, generating a per-package policy that captures legitimate capabilities. Then, we inject obfuscated malicious code, using four Go-specific obfuscation strategies [58]:

- **Plugin**: we compile the malicious functionality as a Go plugin, and load it at runtime.
- **Reflection**: we wrap malicious functions with reflection calls, with dynamic resolution to invoke them.
- **External Binary**: we embed the payload in a separate binary, and invoke it at runtime via `exec` syscall.
- **CGO**: we write the malicious code as C code, and invoke it through Go's foreign function interface (CGO).

Table 7.4. Detection Rate of GoLeash against obfuscated attacks

(a) Malware M2 (data exfiltration)

Project	w/o(%)	plugins(%)	reflection(%)	binary(%)	cgo(%)
k8s	99.77	99.77	99.77	100	99.09
etcd	98.39	98.39	98.39	100	98.39
coredns	100	100	100	100	90
frp	97.87	93.62	93.62	100	76.60
geth	97.59	97.59	97.59	100	96.39
avg.	98.72	97.87	97.87	100.00	92.09

(b) Malware M5 (remote file infiltration)

Project	w/o(%)	plugins (%)	reflection (%)	binaries (%)	cgo(%)
k8s	99.55	99.55	99.55	100	99.32
etcd	98.39	98.39	98.39	100	98.39
coredns	100	100	100	100	95
frp	95.74	95.74	95.74	100	85.11
geth	97.59	97.59	100	96.39	96.39
avg.	98.25	98.25	98.33	99.28	94.44

Each of these obfuscation techniques conceals the malicious injected capabilities behind more convoluted, indirect execution paths.

We reuse the same injection framework described in Section 7.5, but apply the above four evasion strategies to the malicious snippets before injecting them into each exercised package. In this second experiment, we focus on two malicious behaviors (i.e., M2 data exfiltration and M5 remote file infiltration) from the original set, for the sake of brevity. Similar results were obtained for the other types of malware. Each obfuscation variant is injected one at a time into each exercised package across the five Go projects. We then switch GoLeash to enforcement mode and rerun the applications with their workload to measure if GoLeash detects these obfuscated attacks.

Results. Tables 7.4a and 7.4b report detection rates of GoLeash for both unobfuscated malware as in RQ1 (column `w/o`), and their obfuscated counterparts. For each exercised package in the target applications, we generate eight malicious injections (two malware variants and four evasion techniques), resulting in a total of $5,224$ injections across all five applica-

tions.

Despite attempts to conceal malicious capabilities, GoLeash still identifies unauthorized behavior that falls outside a package's generated policy, achieving average detection rates across all projects ranging from 92.09% to 100% for malware M2, and from 94.44% to 99.28% for malware M5, depending on the obfuscation technique. Our experiment shows that the evasion strategies poorly circumvent GoLeash's per-package policies. Malicious accesses are still flagged as suspicious calls whenever they deviate from a package's baseline profile. For example, malware based on CGO still needs to issue system calls for connecting to remote hosts, such as those mapped to the `CONNECT_REMOTE` capability, which remains detectable. Interestingly, in the scenario of external binary evasion, GoLeash's detection rate is slightly higher than in the unobfuscated variant, primarily because this approach introduces the `EXEC` capability, which is absent in most of the baseline profile and immediately triggers an alert.

> GoLeash is highly robust against obfuscated software supply chain attacks, where the malicious code in obfuscated inside a dependency. Despite obfuscation, GoLeash detects the deviations from the security policy.

7.5.3 Performance Overhead

Objective. The goal of this evaluation is to quantify the performance overhead introduced by GoLeash when used in enforcement mode. Recall that the enforcement mode is meant to be used in production. In enforcement mode, GoLeash actively intercepts and potentially blocks system calls from the target application. Since runtime overhead can impact on real-world adoption, it is critical to measure how much additional cost is imposed by GoLeash.

Methodology. For this evaluation, we run the applications using benchmark workloads that offer consistent, high-volume execution, making them better suited for measuring performance metrics. These benchmarks replace the workloads used in RQ1, which were meant to perform end-to-end testing to exercise the several functionalities of the applications.

We consider the same five Go applications, with the following work-

loads. For `etcd`, we configure its official benchmarking tool [243] to send write requests to a running `etcd` cluster. For `frp`, we set up a tunnel to a web server through the frp proxy and generate HTTP traffic using the `wrk` benchmarking tool [241]. For `geth`, we deploy an Ethereum node and submit transaction requests using its benchmarking suite [242]. For `CoreDNS`, we issue DNS queries using `DNSPerf` [88]. For all of these applications, we measure the total *execution time* over 10 repetitions, configuring benchmark for executing 10,000 requests each. For `Kubernetes`, we simulate typical API operations using the `kube-burner` benchmark [162]. In this case, rather than measuring request execution time, we use *pod latency*, that is the time it takes for a pod to reach the `Ready` state after being scheduled. This metric is widely used to benchmark the responsiveness and provisioning performance of Kubernetes clusters.

In addition, for all applications, we measure *syscall latencies* introduced by the kernel probes, averaging over 1,000 traced syscalls per project. Before each repetition, we reset the environment (e.g., by clearing application-generated caches) to ensure consistency.

Table 7.5. Execution Time overhead measurements.

Project	w/o (s)	GoLeash (s)	Overhead (%)
`kubernetes`	17.50 ± 0.56	19.00 ± 0.49	$+8.57\%$
`etcd`	7.09 ± 0.13	7.48 ± 0.10	$+5.5\%$
`coredns`	13.16 ± 0.24	13.74 ± 0.49	$+4.4\%$
`frp`	0.64 ± 0.005	0.79 ± 0.004	$+23.43\%$
`geth`	22.54 ± 1.08	23.63 ± 0.82	$+4.83\%$

Results. Table 7.5 shows the mean and standard deviation of performance measurements, both without (`w/o`) and with GoLeash, along with the relative overhead as percentage. We observe an average overhead of 9.34% in *execution time*, primarily due to kernel-level probes on `sys_enter` and user-space operations such as syscall-capability mapping, dependency attribution, and allowlist inspection. These results align with previous studies on eBPF-based tracing [79]. In addition, we observe an average syscall latency of $3.53ms$ for `Kubernetes`, $3.09ms$ for `etcd`, $2.8ms$ for `CoreDNS`, $3.01ms$ for `frp`, and $3.44ms$ for `geth`, respectively.

> Our measurements show that GoLeash's enforcement mode adds 4-25% execution overhead in production. This overhead is an acceptable cost for the security benefits provided by GoLeash against the state-of-the-art software supply chain attacks.

7.5.4 Comparison with Static Analysis

Objective. Our goal in this experiment is to assess how GoLeash compares to *Capslock* [124]. Capslock is a state-of-the-art static capability analysis tool for Go, developed by Google. Capslock obtains the transitive dependency graph of the application under analysis. Then, for each package used by the application, it infers whether the package can manipulate files, open network connections, or perform certain system-level operations. It scans the source code by means of static analysis, by looking for calls to the Go standard library in the call graph. These capabilities are similar to those used by GoLeash (see Table 7.1), even if at a coarser level of granularity. We investigate whether Capslock's static analysis can effectively detect unobfuscated and obfuscated malicious code, and how it compares to GoLeash's runtime-based detection.

Methodology. Similarly to the previous RQ1 and RQ2, we evaluate Capslock in terms of detection rate. First, we apply Capslock on the original (benign) version of the five applications, and obtain the set of capabilities inferred by Capslock. Then, we apply Capslock on the malicious application variants. For each malicious variant, we get the set of capabilities, which includes the capabilities used by the injected malicious code. If the original and the malicious applications exhibit a different set of capabilities, we conclude that the malicious code has been detected by Capslock. For fair comparison, we compute the detection rate for both GoLeash and Capslock on the same datasets of injected packages from RQ1 and RQ2.

Results. For space limitations, we again focus on two malware types. Tables 7.6a and 7.6b report the absolute differences in detection rates between GoLeash and Capslock, respectively for the M2 and M5 malware types, across the five Go projects. We consider both the cases

Table 7.6. Differences in Detection Rate of GoLeash against CapSlock

(a) Malware M2 (Data Exfiltration)

Project	w/o (%)	plugins (%)	reflection (%)	binaries (%)	cgo (%)
k8s	+76.64	+67.12	+80.95	+64.85	+63.04
etcd	+62.91	+58.07	+69.36	+58.06	+56.45
coredns	+50.00	+50.00	+60.00	+45.00	+20.00
frp	+70.21	+29.79	+70.22	+36.17	+10.64
geth	+62.65	+32.53	+67.47	+33.73	+27.72
avg.	+64.88	+47.90	+69.20	+47.96	+35.97

(b) Malware M5 (Remote File Infiltration)

Project	w/o (%)	plugins (%)	reflection (%)	binaries (%)	cgo (%)
k8s	+74.38	+66.90	+80.73	+64.85	+63.27
etcd	+59.68	+58.07	+69.36	+58.06	+56.45
coredns	+45.00	+50.00	+60.00	+45.00	+25.00
frp	+53.19	+53.19	+72.34	+36.17	+19.15
geth	+56.63	+32.53	+67.47	+33.73	+27.72
avg.	+57.38	+52.54	+69.58	+47.96	+38.72

without (`w/o`) and with obfuscation. Each cell shows the percentage-point difference between GoLeash's dynamic detection results from RQ2 and Capslock's static analysis.

The results shows that GoLeash consistently and significantly outperforms Capslock across all five projects and obfuscation techniques. This performance gap stems from two main limitations of Capslock's analysis. First, Capslock operates at a coarser level of granularity than GoLeash, which causes overlap between broad capabilities attributed to trusted packages and those required by injected malware. As a result, it often fails to distinguish malicious behavior from benign behavior. For example, read and write operations fall in the same capability in Capslock, obscuring distinctions that GoLeash can capture at the syscall level. Second, Capslock's static analysis struggles to precisely reconstruct call graphs, especially in the presence of dynamic features such as plugin loading or reflection. Based on manual inspection of randomly sampled cases, we observed that Capslock occasionally misses even coarse-grained capabilities that are clearly exercised during runtime.

Interestingly, when CGO-based obfuscation is used, GoLeash still outperforms Capslock in every scenario, although the percentage-point advantage is somewhat reduced. This is because CGO variants introduce additional calls to Go standard libraries and foreign function interfaces, which Capslock explicitly flags (e.g., via `CGO_CAPABILITY`). However, Capslock often raises alerts on the presence of these foreign interfaces rather than on the core malicious logic itself (e.g., actual file writes or exfiltration operations).

It is important to note that Capslock can still bring unique benefits to supply chain security. Our approach relies on the actual execution of the target application with a representative workload that triggers its capabilities, which is typically the case of production-grade software. Looking at actual executions enables our solutions to overcome obfuscation and other limitations of static analysis. However, if users cannot find or run such a workload, it is not possible to perform dynamic analysis. In these cases, static analysis can still be used to obtain an approximation of the capabilities used within the application.

> GoLeash's dynamic analysis outperforms Capslock's static analysis. Yet, we note that they are complementary, and that practical defense-in-depth against software supply chain attacks require using both kinds of solution.

7.6 Limitations

We here discuss key design choices and considerations that shape the applicability of GoLeash.

Workload Coverage GoLeash uses dynamic analysis to generate fine-grained, per-package policies based on observed behavior. Its coverage is thus limited by the code exercised during analysis. If (1) rarely executed paths are missed, and (2) those paths use additional capabilities, GoLeash may flag them as violations at runtime (i.e., false positives). This is a known limitation of dynamic analysis [246], not unique to our system. In practice, since we propose using integration tests as workload for policy generation, this problem is mitigated in production-grade software with ro-

bust test suites, as is common in critical cloud systems. Moreover, GoLeash supports the incremental update of security policies, which enables users to conservatively limit capabilities as observed during testing, and later enable capabilities that trigger false positives after more careful scrutiny.

Capability Reuse GoLeash enforces policies by detecting when packages invoke unauthorized capabilities. However, if malware is injected into a permissive module and reuses already-allowed capabilities, it may evade detection under our current model. Still, GoLeash significantly limits such opportunities, as most packages use a narrow set of capabilities, as shown in our experiments. Detecting residual attacks would require deeper inspection of syscall arguments and side effects (e.g., destination IPs, file paths), as explored in prior work [304, 246, 228]. Such analysis is orthogonal to our focus on per-package capability boundaries, and can be integrated into GoLeash.

Policy Usability. GoLeash assumes that capability policies generated in analysis mode are reviewed and approved by human operators (e.g., developers or platform engineers) before deployment. In our experiments, this role is implicitly played by the authors, and we do not include a user study assessing how practitioners interact with GoLeash in practice. In real deployments, there is a risk that overloaded users might approve overly permissive policies (e.g., by accepting all observed capabilities or blindly whitelisting new violations), which would weaken the effectiveness of enforcement and reduce GoLeash to a logging mechanism. This human-in-the-loop aspect is therefore a threat to the validity and external applicability of our results. A systematic user study with engineers and operators, for example, measuring effort, accuracy, and the tendency to over-approve capabilities when using GoLeash in realistic workflows, is an important direction for future work.

Vulnerabilities in third-party software. This work focuses on detecting malicious code introduced via upstream dependencies, a major software supply chain risk. Benign packages may also contain vulnerabilities, exposing dependent applications to attacks. This is another area of interest, covered by tools such as Trivy and Snyk [34, 263], but outside our

scope. GoLeash indirectly mitigates certain vulnerabilities, particularly those enabling remote code execution (RCE), such as command injection and deserialization flaws such as Log4Shell [201] and the recent Ingress-nginx RCE [217]. In such cases, the exploited package would invoke a capability outside its policy, which GoLeash would flag as a violation.

Portability Across Languages and Systems GoLeash is designed for the Go language on Linux, leveraging eBPF tracing and Go-specific features (e.g., package management, symbol tables). However, its core design principles (e.g., stack-based syscall attribution, per-component capability policies), are conceptual and portable to other languages and systems. For instance, similar techniques could apply to Java and Rust, which also support package management and expose symbols and package information in runtime stack traces. Additionally, other operating systems, such as FreeBSD and Windows, also offer production-grade tracing frameworks, such as DTrace [195].

Stripped Binaries GoLeash relies on Go symbol tables to map system calls to their originating packages via stack traces. This works reliably because Go binaries are typically not stripped in production environments [125]. In the rare case of stripped binaries, such as in performance-critical deployments, GoLeash can integrate with tools like `redress` [127] to heuristically recover function boundaries and naming information. While this may reduce attribution precision, enforcement remains effective through hashed stack traces and recovered labels. Therefore, stripped binaries are a deployment consideration, but not a fundamental obstacle for using GoLeash.

7.7 Discussion

GoLeash is a novel solution for mitigating software supply chain attacks in Go. By enforcing runtime least-privilege boundaries at the level of individual Go packages, GoLeash can detect malicious capabilities introduced through compromised dependencies. These fine-grained enforcement mechanisms enable it to expose stealthy behaviors that process-level techniques or static analyses fail to capture. Our evaluation shows that

GoLeash achieves high detection accuracy, remains effective under obfuscation, and incurs acceptable runtime overhead. Beyond its practical impact, GoLeash reinforces the central thesis of this dissertation: attack-surface reduction must occur precisely at the level where complexity and privilege intersect. In the context of software supply chains, this interface lies between trusted and untrusted dependencies, where code reuse, automation, and third-party integration create implicit trust channels. GoLeash operationalizes least-privilege principles at this boundary, turning package-level capabilities into a verifiable, enforceable abstraction. In this perspective, GoLeash extends the dissertation's trajectory from prior chapters: whereas Kubefence secures orchestration APIs and Iris explores hypervisor interaction surfaces, GoLeash targets the interfaces of software composition itself. Together, these systems illustrate a unifying philosophy, systematically reducing attack surfaces by enforcing minimal privilege at each layer where software complexity manifests, from orchestration and virtualization to the supply chain.

Chapter 8

Conclusions

Modern software systems are the product of decades of engineering that prize modularity, openness, and composability. Each abstraction layer, from orchestration and virtualization to language runtimes and supply chains, extends functionality while introducing new interfaces, dependencies, and control channels. These features, though essential for scalability and flexibility, collectively enlarge the software attack surfaces.

This dissertation set out to investigate how such attack surfaces can be systematically reduced. Rather than treating security as an afterthought or a reactive process of patching vulnerabilities, the research pursued a proactive vision: reducing unnecessary exposure at the interfaces where complexity accumulates. Each layer of the software stack, whether the orchestration control plane, the inter-process communication boundary, the hypervisor, or the software supply chain, introduces distinct forms of exposure that cannot be eliminated by generic hardening alone. Instead, each requires enforcement mechanisms that are fine-grained, automated, and resilient to evasion, operating directly on the abstractions that define privilege at that layer.

Through five complementary techniques, that are *KubeFence, FuzzBox, IRIS, GoSurf, and GoLeash*, this dissertation demonstrated that such reduction is both feasible and effective across heterogeneous layers. *KubeFence* applied runtime enforcement to Kubernetes APIs, automatically generating workload-specific least-privilege policies to debloat orchestration operations. *FuzzBox* targeted inter-process communication surfaces,

integrating coverage-guided fuzzing within full-system emulation to explore opaque components within industrial binaries. *IRIS* refined this approach for hardware-assisted virtualization, enabling high-fidelity fuzzing through record-and-replay of guest–hypervisor interactions. *GoSurf*, in turn, addressed the static dimension of software supply-chain security by introducing the first Go-specific taxonomy of attack vectors and a static analysis tool to identify latent risks before deployment. Finally, *GoLeash* implemented runtime enforcement by applying least-privilege principles at the level of Go packages, dynamically constraining their system-call capabilities during execution.

Taken together, these contributions form a coherent trajectory from the outermost control surfaces to the innermost layers of software composition. They collectively show that interfaces define the structural foundations of modern attack surfaces. Each technique operates at a natural enforcement boundary: Kubernetes APIs as control interfaces, IPC channels as communication boundaries, VM exits as virtualization interfaces, and package imports as compositional boundaries. By focusing reduction efforts at these precise points, the dissertation demonstrates that it is possible to achieve localized least privilege without sacrificing the functionalities and performance that make modern infrastructures possible.

The developed techniques also reveal the complementarity between static and dynamic perspectives. Static methods, such as GoSurf's taxonomy-driven analysis, expose the potential execution paths through which malicious behaviors can manifest, while dynamic mechanisms such as GoLeash, KubeFence, or IRIS constrain and verify behavior at runtime. This synergy transforms attack-surface reduction into a continuous process rather than a one-time configuration task: static analysis informs enforcement, and runtime observation feeds back into refined policies. In this perspective, automation emerges as an indispensable enabler of security. Manual policy engineering, seed generation, or configuration tuning cannot keep pace with the complexity and dynamism of today's infrastructures. The tools introduced in this dissertation, including automatic policy inference in KubeFence, deterministic state replay in IRIS, dynamic capability tracing in GoLeash, demonstrate that scalable reduction is attainable only when automation is coupled with precision.

Naturally, this research operates within certain boundaries. Each sys-

tem assumes some degree of visibility into interfaces (system calls, APIs, or dependency graphs) that may not be available in closed or proprietary environments. Fine-grained enforcement also incurs overheads, and although prototypes such as GoLeash and KubeFence demonstrate acceptable performance, scaling these mechanisms to industrial deployments will require adaptive optimization and tighter integration into orchestration pipelines. Another challenge concerns evolution: APIs, kernels, and dependencies change over time, and so must the policies that govern them. Sustaining least-privilege enforcement in the face of such evolution calls for self-updating and context-aware policy mechanisms, which remain open research problems. Finally, while the proposed systems were validated experimentally on realistic targets, industrial adoption will require formal standardization, broader interoperability, and usability studies to balance security with operational practicality.

The limitations identified across the proposed systems naturally open several avenues for future research. These include extending workload-aware enforcement beyond specific deployment formats (e.g., beyond Helm), improving emulation fidelity and automation in full-system fuzzing, scaling record-and-replay fuzzing to multi-core and heterogeneous virtualization platforms, and conducting large-scale user studies to assess policy usability and operational impact. Further directions include integrating deeper semantic inspection of capabilities, extending package-level enforcement to additional languages and operating systems, and developing self-adaptive policy mechanisms that evolve with changing APIs, kernels, and dependencies.

In conclusion, this dissertation advances the state of the art in systematic attack-surface analysis and reduction and embodies a philosophy of *security through reduction*: securing modern infrastructures not by adding defensive layers, but by precisely removing unnecessary exposure where complexity and privilege meet.

Bibliography

[1] BlackBerry Limited., Are Hypervisors the Answer to the Coming silicon Shortages? https://blackberry.qnx.com/content/dam/blackberry-com/Documents/pdf/BlackBerry_QNX_Hypervisor_WhitePaper_22April2021_FINAL.pdf.

[2] Fuzzable - Framework for Automating Fuzzable Target Discovery with Static Analysis. https://github.com/ex0dus-0x/fuzzable.

[3] Volatility. https://www.volatilityfoundation.org/.

[4] Wind RiverVxWorks MILS 3.0 BSP for WR SBC8548. https://bsp.windriver.com/bsps/1573.

[5] google - Syzkaller. https://github.com/google/syzkaller, 2015.

[6] Robert, Swiecki - Honggfuzz. http://code.google.com/p/honggfuzz, 2016.

[7] M. Zalewski - American fuzzy lop, author = , 2017. http://lcamtuf.coredump.cx/afl/.

[8] Heuse, Marc - AFL-DynamoRIO. https://github.com/vanhauser-thc/afl-dynamorio, 2018.

[9] talos-vulndev - AFL-DynInst. https://github.com/talos-vulndev/afl-dyninst, 2018.

[10] QEMU AdaCore. https://github.com/AdaCore/qemu/blob/qemu-stable-4.0.0/hw/ppc/p2010rdb.c, 2019.

[11] Heuse, Marc - AFL-PIN. https://github.com/vanhauser-thc/afl-pin, 2020.

[12] sendmail. https://github.com/mykter/afl-training/tree/main/challenges/sendmail/1305, 2020.

[13] json-parser. https://github.com/json-parser/json-parser, 2022.

[14] TinyExpr. https://github.com/codeplea/tinyexpr, 2022.

[15] GNU GCC. https://gcc.gnu.org/, 2023.

[16] GNU Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2023.

[17] IBM - PowerPC register usage conventions. https://www.ibm.com/docs/en/aix/7.2?topic=overview-register-usage-conventions, 2023.

[18] QEMU Architectures Support. https://wiki.qemu.org/Documentation/Platforms, 2023.

[19] QEMU TCG Plugins. https://github.com/qemu/qemu/blob/master/docs/devel/tcg-plugins.rst, 2023.

[20] Wind River - VxWorks Workbench. https://docs.windriver.com/bundle/Workbench_4_Getting_Started_OpenVersion_23_03/page/elx1502803804955.html, 2023.

[21] Dlink - DCS-932L Firmware. https://support.dlink.com.au/download/download.aspx?product=DCS-932L, 2024.

[22] Port Swigger - BurpSuite. https://portswigger.net/burp, 2024.

[23] Tenda - AC15 Firmware. https://www.tendacn.com/it/product/download/a15.html, 2024.

[24] Trendnet - TEW-651BR Firmware. https://www.trendnet.com/support/support-detail.asp?prod=190_TEW-651BR, 2024.

[25] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. Cage4Deno: A fine-grained sandbox for Deno subprocesses. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, pages 149–162, 2023.

[26] Marco Abbadini, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. NatiSand: Native code sandboxing for JavaScript runtimes. In *26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 639–653, 2023.

[27] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel technology journal*, 10(3), 2006.

[28] Airlines Electronic Engineering Committee. ARINC 811: Commercial aircraft information security concepts of operation and process framework. *Aeronautical Radio, Inc*, 2005.

[29] Akamai. Can't Be Contained: Finding a Command Injection Vulnerability in Kubernetes. `https://www.akamai.com/blog/security-research/kubernetes-critical-vulnerability-command-injection`, 2023.

[30] Jim Alves-Foss, Paul W Oman, Carol Taylor, and W Scott Harrison. The MILS architecture for high-assurance embedded systems. *International journal of embedded systems*, 2006.

[31] AMD Technology. AMD64 Architecture Programmer's Manual Volume 2: System Programming.

[32] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. Virtual CPU validation. *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[33] Paschal C Amusuo, Kyle A Robinson, Tanmay Singla, Huiyun Peng, Aravind Machiry, Santiago Torres-Arias, Laurent Simon, and James C Davis. Ztd_{JAVA}: Mitigating software supply chain vulnerabilities via zero-trust dependencies. *IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025.

[34] Aqua Security. Trivy: Open source vulnerability and misconfiguration scanner, 2025. Accessed: 2025-04-12.

[35] Artifact Hub. MLflow Operator. `https://artifacthub.io/packages/helm/community-charts/mlflow`, 2024.

[36] Artifact Hub. Nginx Operator. `https://artifacthub.io/packages/helm/bitnami/nginx`, 2024.

[37] Artifact Hub. PostgreSQL Operator. `https://artifacthub.io/packages/helm/bitnami/postgresql`, 2024.

[38] Artifact Hub. RabbitMQ Operator. `https://artifacthub.io/packages/helm/bitnami/rabbitmq`, 2024.

[39] Artifact Hub. SonarQube Operator. `https://artifacthub.io/packages/helm/openshift-bootstraps/sonarqube`, 2024.

[40] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Checking security properties of cloud service rest apis. In *IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 387–397, 2020.

[41] Anonymous Author(s). Replication Package for "GoLeash: Mitigating GoLang Software Supply Chain Attack with Runtime Polocy Enforcement". https://figshare.com/s/c4844c75354df7e94b55, 2025. Accessed: 2025-04-15.

[42] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. FUDGE: Fuzz driver generation at scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[43] VulnCheck Jacob Baines. Hijackable Go Module Repositories. https://vulncheck.com/blog/go-repojacking, December 2023. Online; accessed 9 June 2024.

[44] Julian Bangert and Nickolai Zeldovich. Nail: A practical tool for parsing and generating data formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 615–628, 2014.

[45] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[46] Marco Barletta, Luigi De Simone, Raffaele Della Corte, and Catello Di Martino. Failover timing analysis in orchestrating container-based critical applications. In *19th European Dependable Computing Conference (EDCC)*, pages 81–84, 2024.

[47] Christoph Baumann, Thorsten Bormer, Holger Blasum, and Sergey Tverdyshev. Proving memory separation in a microkernel by code level verification. In *Proc. ISORC*, pages 25–32, 2011.

[48] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*. Califor-nia, USA, 2005.

[49] Benjamin Grap, and Manoj Ahuje. CrowdStrike discovers first ever dero cryptojacking campaign targeting Kubernetes. https://www.crowdstrike.com/en-us/blog/crowdstrike-discovers-first-ever-dero-cryptojacking-campaign-targeting-kubernet 2024.

[50] William Blair, Frederico Araujo, Teryl Taylor, and Jiyong Jang. Automated synthesis of effect graph policies for microservice-aware stateful system call specialization. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4554–4572. IEEE, 2024.

[51] Agathe Blaise and Filippo Rebecchi. Stay at the Helm: secure Kubernetes deployments via graph generation and attack reconstruction. In *IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 59–69, 2022.

[52] Dibyendu Brinto Bose, Akond Rahman, and Shazibul Islam Shamim. 'under-reported' security defects in Kubernetes manifests. In *IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, pages 9–12, 2021.

[53] Kirill Boychenko. Go Supply Chain Attack: Malicious Package Exploits Go Module Proxy Caching for Persistence. `https://socket.dev/blog/malicious-package-exploits-go-module-proxy-caching-for-persistence`, 2025.

[54] Kirill Boychenko. Typosquatted Go Packages Deliver Malware Loader Targeting Linux and macOS Systems. `https://socket.dev/blog/typosquatted-go-packages-deliver-malware-loader`, 2025.

[55] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. Morphuzz: Bending (Input) Space to Fuzz Virtual Devices. In *USENIX Security Symposium*, 2022.

[56] Charly Castes, François Costa, Nate Foster, Thomas Bourgeat, and Edouard Bugnion. Lightweight hypervisor verification: Putting the hardware burger on a diet. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, pages 27–33, 2025.

[57] Frederico Cerveira, Raul Barbosa, Henrique Madeira, and Filipe Araújo. The effects of soft errors and mitigation strategies for virtualization servers. *IEEE TCC*, 2020.

[58] Carmine Cesarano, Vivi Andersson, Roberto Natella, and Martin Monperrus. Gosurf: Identifying software supply chain attack vectors in go. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 33–42, 2023.

[59] Carmine Cesarano, Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Giorgio Farina. Iris: a record and replay framework to enable hardware-assisted virtualization fuzzing. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 389–401. IEEE, 2023.

[60] Carmine Cesarano, Alessio Foggia, Gianluca Roscigno, Luca Andreani, and Roberto Natella. Genio: Synergizing edge computing with optical network infrastructures. *IEEE Communications Magazine*, 2025.

[61] Carmine Cesarano, Martin Monperrus, and Roberto Natella. Goleash: Mitigating golang software supply chain attacks with runtime policy enforcement. *arXiv preprint arXiv:2505.11016*, 2025.

[62] Carmine Cesarano and Roberto Natella. Fuzzbox: Blending fuzzing into emulation for binary-only embedded targets. *arXiv preprint arXiv:2509.05643*, 2025. Accepted for publication in Springer Cybersecurity Journal.

[63] Carmine Cesarano and Roberto Natella. Kubefence: Security hardening of the kubernetes attack surface. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 497–510, 2025.

[64] Checkmarkx. KIKS. `https://kics.io`, 2024.

[65] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[66] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019.

[67] Alessandro Cilardo, Marcello Cinque, Luigi De Simone, and Nicola Mazzocca. Virtualization Over Multiprocessor Systems-on-Chip: An Enabling Paradigm for the Industrial Internet of Things. *Computer*, 55(10):35–47, 2022.

[68] cilium. bpf2go, 2025. Accessed: 2025-04-12.

[69] Cilium. ebpf-based networking, observability, security, 2025. Accessed: 2025-04-12.

[70] Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Generation Computer Systems*, 2021.

[71] Cloud Native Community Group. KubeArmor. `https://kubearmor.io`, 2024.

[72] Cloud Native Computing Foundation. Annual survey. `https://www.cncf.io/reports/cncf-annual-survey-2023/`, 2024.

[73] Cloud Native Computing Foundation. Artifact Hub. `https://artifacthub.io/`, 2024.

[74] CloudSecDocs. Kubernetes Threat Model. `https://cloudsecdocs.com/containers/theory/threats/k8s_threat_model/#threat-actors`, 2024.

[75] CoreDNS Authors. Coredns: Dns server that chains plugins, 2025. Accessed: 2025-04-12.

[76] Aldo Cortesi, Maximilian Hils, and Thomas Kriechbaumer. mitmproxy. `https://mitmproxy.org/`, 2024.

[77] D. Costa, S. Mujahid, R. Abdalkareem, and E. Shihab. Breaking type safety in go: An empirical study on the usage of the unsafe package. *IEEE Transactions on Software Engineering*, 48(07):2277–2294, jul 2022.

[78] Domenico Cotroneo, Luigi De Simone, and Roberto Natella. Timing covert channel analysis of the vxworks mils embedded hypervisor under the common criteria security certification. *Elsevier COSE*, 106:102307, 2021.

[79] Milo Craun, Khizar Hussain, Uddhav Gautam, Zhengjie Ji, Tanuj Rao, and Dan Williams. Eliminating ebpf tracing overhead on untraced processes. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, pages 16–22, 2024.

[80] Synopsys Cybersecurity Research Center (CyRC). Open Source Security and Risk Analysis (OSSRA) Report, 2024.

[81] C. Dall, S. Li, J. T. Lim, J. Nieh, and G. Koloventzos. ARM virtualization: performance and architectural implications. In *Annual International Symposium on Computer Architecture*, pages 304–316. IEEE, 2016.

[82] Datadog. Guarddog: Cli tool to identify malicious pypi, npm packages, 2024. Accessed: 2025-04-12.

[83] Datadog. Malicious software packages dataset, 2025. Accessed: 2025-04-12.

[84] Giorgio Dell'Immagine, Jacopo Soldani, and Antonio Brogi. KubeHound: Detecting Microservices' Security Smells in Kubernetes Deployments. *Future Internet*, 15(7), 2023.

[85] Go Dev. Case Studies. `https://go.dev/solutions/case-studies`, 2025.

[86] Edel Díaz, Raúl Mateos, Emilio J Bueno, and Rubén Nieto. Enabling parallelized-qemu for hardware/software co-simulation virtual platforms. *Electronics*, 2021.

[87] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[88] DNS-OARC. dnsperf: Dns performance testing tools, 2025. Accessed: 2025-04-12.

[89] Docker Inc. Docker: Empowering app development for developers. `https://www.docker.com`, 2024. Accessed: 2025-04-11.

[90] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016.

[91] Prashanth Varma Dommaraju. *Erroneous Kubernetes Object Generation using Structure-aware Fuzzing*. PhD thesis, Universiteit van Amsterdam, 2024.

[92] Xuechao Du, Andong Chen, Boyuan He, Hao Chen, Fan Zhang, and Yan Chen. AflIot: Fuzzing on linux-based IoT device with binary-level instrumentation. *Computers & Security*, 122:102889, 2022.

[93] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08*, 2008.

[94] eBPF Foundation. What is ebpf? `https://ebpf.io/what-is-ebpf/`, 2024. Accessed: 2025-04-11.

[95] eBPF.io authors. eBPF. `https://ebpf.io/`, 2025.

[96] Jake Edge. A seccomp overview. `https://lwn.net/Articles/656307/`, 2015.

[97] Frank Effenberger and Tarek S El-Bawab. Passive optical networks (PONs): past, present, and future. *Optical Switching and Networking*, 6(3):143–150, 2009.

[98] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing embedded systems using debug interfaces. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*., 2023.

[99] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecurity*, 5(1):18, 2022.

[100] etcd Authors. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. `https://etcd.io`, 2024. Accessed: 2025-04-11.

[101] Ethereum Foundation. go-ethereum: Go implementation of the ethereum protocol, 2025. Accessed: 2025-04-12.

[102] European Commission. EU Cyber Resilience Act. `https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act`, 2024.

[103] Fairwinds. Polaris. `https://www.fairwinds.com/polaris`, 2024.

[104] Falco. Sysdig: Detect security threats in real time, 2025. Accessed: 2025-04-12.

[105] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. Sok: Enabling security analyses of embedded systems via rehosting. In *Proceedings of the 2021 ACM Asia conference on computer and communications security*, 2021.

[106] fatedier. frp: A fast reverse proxy to help you expose a local server behind a nat or firewall to the internet, 2025. Accessed: 2025-04-12.

[107] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[108] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1334–1346. IEEE Press, 2021.

[109] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[110] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[111] Pedro Fonseca, Xi Wang, and Arvind Krishnamurthy. MultiNyx: a multi-level abstraction framework for systematic analysis of hypervisors. *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[112] Leo Freitas and John McDermott. Formal methods for security in the xenon hypervisor. *International journal on software tools for technology transfer*, 13(5):463–489, 2011.

[113] Jian Gao, Yiwen Xu, Yu Jiang, Zhe Liu, Wanli Chang, Xun Jiao, and Jiaguang Sun. Em-fuzz: Augmented firmware fuzzing via memory checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[114] Michael Gasch. Go 1.20 Coverage Profiling Support for Kubernetes Apps. `https://www.mgasch.com/2023/02/go-e2e/`, 2024.

[115] Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[116] Sijia Geng, Yuekang Li, Yunlan Du, Jun Xu, Yang Liu, and Bing Mao. An empirical study on benchmarks of artificial software vulnerabilities. *arXiv preprint arXiv:2003.09561*, 2020.

[117] Adrien Ghosn, Marios Kogias, Mathias Payer, James R Larus, and Edouard Bugnion. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 255–267, 2021.

[118] GitHub. GitHub Advisory Database. `https://github.com/advisories`, 2024.

[119] GitHub. Github, 2025. Accessed April 12, 2025.

[120] github user. CasaOS. `https://github.com/IceWhaleTech/CasaOS/blob/main/main.go`, 2024. Online; accessed 9 June 2024.

[121] Go. Vulnerability database. `https://pkg.go.dev/vuln/`, 2024. Online; accessed 20 June 2024; Language version go1.22.

[122] Go Team. Go standard library - go packages, 2023. Accessed April 2025.

[123] Go Team. Go standard library - the runtime package, 2023. Accessed April 2025.

[124] Google. Capslock: Capability analysis cli for go packages, 2025. Accessed: 2025-04-12.

[125] Google Cloud Blog. Ready, Set, Go — Golang Internals and Symbol Recovery. `https://cloud.google.com/blog/topics/threat-intelligence/golang-internals-symbol-recovery/`, 2022.

[126] Nuwan Goonasekera, William Caelli, and Colin Fidge. Libvm: an architecture for shared library sandboxing. *Software: Practice and Experience*, 45(12):1597–1617, 2015.

[127] goretk. Redress: A tool for analyzing stripped go binaries, 2025. Accessed: 2025-04-12.

[128] Yue Gu, Xin Tan, Yuan Zhang, Siyan Gao, and Min Yang. EPScan: Automated Detection of Excessive RBAC Permissions in Kubernetes Applications. In *IEEE Symposium on Security and Privacy (SP)*, 2025.

[129] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. {FATE} and {DESTINI}: A framework for cloud recovery testing. In *Proc. USENIX NSDI*, 2011.

[130] gVisor Team. gvisor: The container security platform. `https://gvisor.dev/`, 2024. Accessed: 2025-04-11.

[131] Alex Handy. Build Your Kubernetes Operator with the Right Tool. `https://www.redhat.com/en/blog/build-your-kubernetes-operator-with-the-right-tool`, 2021.

[132] Mubin Ul Haque, M Mehdi Kholoosi, and M Ali Babar. KGSecConfig: A Knowledge Graph Based Approach for Secured Container Orchestrator Configuration. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 420–431, 2022.

[133] Hashicorp. Terraform. `https://github.com/hashicorp/terraform/blob/8e4d4663fe37bbae3c0dc1c0bb20b6bcb17b637c/Makefile#L5`, 2024. Online; accessed 9 June 2024.

[134] HashiCorp. Terraform by hashicorp. `https://www.terraform.io`, 2024. Accessed: 2025-04-11.

[135] Hashicorp. Vault. `https://github.com/hashicorp/vault/blob/f7c16796ed2482b5a595b5d89673e4ec5ff8e7fe/Makefile#L162`, 2024. Online; accessed 9 June 2024.

[136] HashiCorp. Nomad. `https://developer.hashicorp.com/nomad`, 2025.

[137] Yacine Hebbal, Sylvie Laniepce, and Jean-Marc Menaud. Virtual Machine Introspection: Techniques and Applications. *2015 10th International Conference on Availability, Reliability and Security*, pages 676–685, 2015.

[138] Constance L Heitmeyer, Myla Archer, Elizabeth I Leonard, and John McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proc. CCS*, pages 346–355, 2006.

[139] Andrew Henderson, Heng Yin, Guang yao Jin, Hao Han, and Hongmei Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *RAID*, 2017.

[140] Sören Henning, Benedikt Wetzel, and Wilhelm Hasselbring. Reproducible benchmarking of cloud-native applications with the kubernetes operator pattern. ″″, 2021.

[141] CO Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide. *Denver,[2006]*, 2022.

[142] Mohammad Majharul Islam and Abdullah Muzahid. Characterizing real world bugs causing sequential consistency violations. In *Proceedings of the 5th USENIX Conference on Hot Topics in Parallelism*, HotPar'13, page 8, USA, 2013. USENIX Association.

[143] ISO. Road vehicles – Functional safety, 2011.

[144] Isovalent. ebpf-based networking, security, and observability, 2025. Accessed: 2025-04-12.

[145] Damir Isovic and Gerhard Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *Proceedings 21st IEEE Real-Time Systems Symposium*. IEEE, 2000.

[146] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium 20*, 2020.

[147] Istio. istio. `https://github.com/istio/istio/blob/8d200be6d52283c806dfce37ae2241efa915f8fd/Makefile.core.mk#L313`, 2024. Online; accessed 9 June 2024.

[148] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Rest web service maintenance through api policy enforcement. Technical report, UC Santa Barbara tech report, 2014.

[149] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. UTopia: Automatic generation of fuzz driver using unit tests. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023.

[150] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium 21*, 2021.

[151] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Symposium on Operating Systems Principles*, 2005.

[152] Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. Prefail: A programmable tool for multiple-failure injection. In *Proc. ACM OOPSLA*, pages 171–188, 2011.

[153] Hugo Kermabon-Bobinnec, Mahmood Gholipourchoubeh, Sima Bagheri, Suryadipta Majumdar, Yosr Jarraya, Makan Pourzandi, and Lingyu Wang. Prospec: Proactive security policy enforcement for containers. In *12th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 155–166, 2022.

[154] Md Shohel Khan, Rubaiyat Sha Fardin Siam, and Muhammad Abdullah Adnan. A framework for checking and mitigating the security vulnerabilities of cloud service restful apis. *Service Oriented Computing and Applications*, pages 1–22, 2024.

[155] Raphaël Khoury and Nadia Tawbi. Which security policies are enforceable by runtime monitors? a survey. *Computer Science Review*, 6(1):27–45, 2012.

[156] Juhwan Kim, Jihyeon Yu, Hyunwook Kim, Fayozbek Rustamov, and Joobeom Yun. Firm-cov: high-coverage greybox fuzzing for iot firmware via optimized process emulation. *IEEE Access*, 9:101627–101642, 2021.

[157] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Proceedings of the 36th Annual Computer Security Applications Conference*, pages 733–745, 2020.

[158] Hirokuni Kitahara, Kugamoorthy Gajananan, and Yuji Watanabe. Highly-scalable container integrity monitoring for large-scale Kubernetes cluster. In *IEEE International Conference on Big Data (Big Data)*, pages 449–454, 2020.

[159] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proc. Linux Symp.*, volume 1, pages 225–230, 2007.

[160] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[161] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[162] Kube-Burner Contributors. kube-burner: Kubernetes performance and scale test orchestration framework, 2025. Accessed: 2025-04-12.

[163] Kubernetes. CHANGELOG v1.29. https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.29.md, 2024. Online; accessed 21 June 2024.

[164] Kubernetes. Kubernetes scalability and performance SLIs/SLOs. https://github.com/kubernetes/community/blob/master/sig-scalability/slos/slos.md, 2024.

[165] Kubernetes. Official CVE Feed. https://kubernetes.io/docs/reference/issues-security/official-cve-feed, 2024.

[166] Kubernetes. Operator Pattern. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, 2024.

[167] Kubernetes. Using RBAC Authorization. https://kubernetes.io/docs/reference/access-authn-authz/rbac/, 2024.

[168] Kubernetes Authors. Kubernetes: Production-grade container orchestration. https://kubernetes.io, 2024. Accessed: 2025-04-11.

[169] Kubernetes blog. Fixing the Subpath Volume Vulnerability in Kubernetes. https://kubernetes.io/blog/2018/04/04/fixing-subpath-volume-vulnerability, 2018.

[170] Kubernetes project. CRD Conversion Webhook e2e test. https://github.com/kubernetes/kubernetes/blob/master/test/e2e/apimachinery/crd_conversion_webhook.go, 2024.

[171] Kubernetes project. Kubernetes e2e tests. https://github.com/kubernetes/kubernetes/tree/master/test/e2e, 2024.

[172] Kubernetes project. Kubernetes User Case Studies. https://kubernetes.io/case-studies/, 2024.

[173] Kubernetes project. Pod security standards. https://kubernetes.io/docs/concepts/security/pod-security-standards, 2024.

[174] Kubescape. Behavioral Cloud Application Detection & Response. https://www.armosec.io/platform/cloud-detection-and-response/, 2024.

[175] Reversing Labs. Update: IconBurst npm software supply chain attack grabs data from apps and websites, 2022. Online; accessed 9 June 2024.

[176] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526. IEEE, 2023.

[177] Piergiorgio Ladisa, Merve Sahin, Serena Elisa Ponta, Marco Rosa, Matias Martinez, and Olivier Barais. The hitchhiker's guide to malicious third-party dependencies. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 65–74, 2023.

[178] Giacomo Lanciano, Manuel Stein, Volker Hilt, Tommaso Cucinotta, et al. Analyzing Declarative Deployment Code with Large Language Models. *CLOSER*, 2023:289–296, 2023.

[179] J. Lauinger, L. Baumgartner, A. Wickert, and M. Mezini. Uncovering the hidden dangers: Finding unsafe go code in the wild. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 410–417, Los Alamitos, CA, USA, jan 2020. IEEE Computer Society.

[180] Marianna Lezzi, Mariangela Lazoi, and Angelo Corallo. Cybersecurity for Industry 4.0 in the current literature: A reference framework. *Computers in Industry*, 103:97–110, 2018.

[181] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified linux kvm hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1782–1799. IEEE, 2021.

[182] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. μafl: non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[183] Jordan Liggitt. audit2rbac. `https://github.com/liggitt/audit2rbac`, 2024.

[184] Dexin Liu, Yue Xiao, Chaoqi Zhang, Kaitao Xie, Xiaolong Bai, Shikun Zhang, and Luyi Xing. {iHunter}: Hunting privacy violations at scale in the software supply chain on {iOS}. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5663–5680, 2024.

[185] Guannan Liu, Xing Gao, Haining Wang, and Kun Sun. Exploring the unchartered space of container registry typosquatting. In *31st USENIX Security Symposium (USENIX Security)*, pages 35–51, 2022.

[186] Qiang Liu, Cen Zhang, Lin Ma, Muhui Jiang, Yajin Zhou, Lei Wu, Wenbo Shen, Xiapu Luo, Yang Liu, and Kui Ren. Firmguide: Boosting the capability of rehosting embedded linux kernels through model-guided kernel execution. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021.

[187] Yang Luo, Hongbo Zhou, Qingni Shen, Anbang Ruan, and Zhonghai Wu. Restpl: Towards a request-oriented policy language for arbitrary restful apis. In *IEEE International Conference on Web Services (ICWS)*, pages 666–671, 2016.

[188] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[189] Mailing list. Xen Memory De-duplication.

[190] Ehud Malul, Yair Meidan, Dudu Mimran, Yuval Elovici, and Asaf Shabtai. GenKubeSec: LLM-Based Kubernetes Misconfiguration Detection, Localization, Reasoning, and Remediation. *arXiv preprint arXiv:2405.19954*, 2024.

[191] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.

[192] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. Parser-directed fuzzing. In *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, pages 548–560, 2019.

[193] Spencer Michaels and Jeff Dileo. Assessing unikernel security. Technical report, Technical report, NCC group, 2019.

[194] Microsoft. Anatomy of a modern attack surface. `https://www.microsoft.com/en-us/security/security-insider/threat-landscape/anatomy-of-a-modern-attack-surface`, 2023. Accessed: 2025-09-02.

[195] Microsoft. Dtrace on windows, 2025. Accessed: 2025-04-12.

[196] Francesco Minna, Fabio Massacci, and Katja Tuma. Analyzing and Mitigating (with LLMs) the Security Misconfigurations of Helm Charts from Artifact Hub. *arXiv preprint arXiv:2403.09537*, 2024.

[197] MITRE. Cwe-441: Unintended proxy or intermediary ('confused deputy'), 2025. Accessed: 2025-04-12.

[198] Borja Molina-Coronado, Antonio Ruggia, Usue Mori, Alessio Merlo, Alexander Mendiburu, and Jose Miguel-Alonso. Light up that droid! on the effectiveness of static analysis features against app obfuscation for android malware detection. *Journal of Network and Computer Applications*, 235:104094, 2025.

[199] Sara Moshtari, Ahmet Okutan, and Mehdi Mirakhorli. A grounded theory based approach to characterize software attack surfaces. In *Proceedings of the 44th International Conference on Software Engineering*, pages 13–24, 2022.

[200] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. MundoFuzz: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference. In *USENIX Security Symposium*, 2022.

[201] National Institute of Standards and Technology (NIST). Cve-2021-44228: Apache log4j2 remote code execution vulnerability, 2025. Accessed: 2025-04-12.

[202] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), 2006.

[203] K Netkachova, K Müller, M Paulitsch, and RE Bloomfield. Security-informed safety case approach to analysing mils systems. In *International Workshop on MILS: Architecture and Assurance for Secure Systems*, 2015.

[204] Henry Neugass, Geoffrey Espin, Hidefume Nunoe, Ralph Thomas, and David Wilner. Vxworks: an interactive development environment and real-time kernel for gmicro. In *Eighth TRON Project Symposium*. IEEE Computer Society, 1991.

[205] NIST. CVE-2010-0435.

[206] NIST. CVE-2011-1936.

[207] NIST. CVE-2020-2732.

[208] NSA, CISA. Kubernetes Hardening Guide. Technical report, NSA, CISA, 2022.

[209] NVD. CVE-2017-1002101. `https://nvd.nist.gov/vuln/detail/cve-2017-1002101`, 2024.

[210] NVD. CVE-2019-11253. `https://nvd.nist.gov/vuln/detail/cve-2019-11253`, 2024.

[211] NVD. CVE-2020-15257. `https://nvd.nist.gov/vuln/detail/cve-2020-15257`, 2024.

[212] NVD. CVE-2020-8554. `https://nvd.nist.gov/vuln/detail/cve-2020-8554`, 2024.

[213] NVD. CVE-2021-21334. `https://nvd.nist.gov/vuln/detail/cve-2021-21334`, 2024.

[214] NVD. CVE-2021-25741. https://nvd.nist.gov/vuln/detail/cve-2021-25741, 2024.

[215] NVD. CVE-2023-2431. https://nvd.nist.gov/vuln/detail/cve-2023-2431, 2024.

[216] NVD. CVE-2023-3676. https://nvd.nist.gov/vuln/detail/cve-2023-3676, 2024.

[217] Nir OhfeldSasson. Ingressnightmare: Cve-2025-1974 - 9.8 critical unauthenticated remote code execution vulnerabilities in ingress nginx, 2025. Accessed: 2025-04-12.

[218] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 23–43. Springer, 2020.

[219] Marc Ohm, Timo Pohl, and Felix Boes. You Can Run But You Can't Hide: Runtime Protection Against Malicious Package Updates For Node.js. *arXiv preprint arXiv:2305.19760*, 2023.

[220] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[221] Open Source Insights Team. Open source insights, 2025. Accessed April 12, 2025.

[222] Vittorio Orbinato, Marco Carlo Feliciano, Domenico Cotroneo, and Roberto Natella. Laccolith: Hypervisor-based adversary emulation with anti-detection. *IEEE Transactions on Dependable and Secure Computing*, 2024.

[223] Owasp. Zap. https://www.zaproxy.org/, 2024.

[224] Go Packages. go vet package. https://pkg.go.dev/cmd/vet, 2024. Online; accessed 20 June 2024.

[225] Go Packages. gosec package. https://pkg.go.dev/github.com/securego/gosec/v2, 2024. Online; accessed 20 June 2024.

[226] Go Packages. govulncheck. https://pkg.go.dev/golang.org/x/vuln/cmd/govulncheck, 2024. Online; accessed 20 June 2024; Language version go1.22.

[227] Go Packages. m-mizutani/goast. https://pkg.go.dev/github.com/m-mizutani/goast, 2024. Online; accessed 20 June 2024.

[228] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, 2020.

[229] Palo Alto Networks. Protecting Against an Unfixed Kubernetes MITM Vulnerability. `https://unit42.paloaltonetworks.com/cve-2020-8554/`, 2020.

[230] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[231] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. Probabilistic naming of functions in stripped binaries. In *Annual Computer Security Applications Conference*, 2020.

[232] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020.

[233] Rob Pike. Go at Google: Language Design in the Service of Software Engineering. `https://go.dev/talks/2012/splash.article`, 2012. Online; accessed 27 May 2024.

[234] Rob Pike. Generating code - The Go Programming Language. `https://go.dev/blog/generate`, 2014. Online; accessed 29 May 2024.

[235] Rob Pike. The Design of the Go Assembler, 2016. Presented at Gophercon. Online; accessed 12 June 2024.

[236] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. In *CACM*, 1973.

[237] Prisma Cloud. Checkov. `https://www.checkov.io`, 2024.

[238] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.*, 32(4), May 2023.

[239] Red Hat. OperatorHub. `https://operatorhub.io/`, 2024.

[240] Red Hat. OpenShift. https://www.redhat.com/it/technologies/cloud-computing/openshift, 2025.

[241] Github repo. wg/wrk: Modern HTTP Benchmarking tool. https://github.com/wg/wrk, 2015.

[242] Github repo. Ethereum Benchmark. https://github.com/OBrezhniev/ethereum-benchmark, 2019.

[243] Github repo. etcd - benchmark. https://github.com/etcd-io/etcd/blob/main/tools/benchmark/README.md, 2024.

[244] Michen Riksen. Finding Evil Go Packages. https://michenriksen.com/archive/blog/finding-evil-go-packages/, 2025. Accessed: 2025-04-15.

[245] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, Jonathan M Smith, Andre DeHon, et al. Mscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 296–311, 2021.

[246] Maryam Rostamipoor, Seyedhamed Ghavamnia, and Michalis Polychronakis. Confine: Fine-grained system call filtering for container attack surface reduction. *Computers & Security*, 132, 2023.

[247] RTCA. DO-178C Software Considerations in Airborne Systems and Equipment Certification *Requirements and Technical Concepts for Aviation*, 1992.

[248] John M Rushby. Design and verification of secure systems. *ACM SIGOPS Operating Systems Review*, 15(5):12–21, 1981.

[249] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[250] Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. An in-depth study of java deserialization remote-code execution exploits and vulnerabilities. *ACM Transactions on Software Engineering and Methodology*, 32(1):1–45, 2023.

[251] Sergej Schumilo, Cornelius Aschermann, Ali Reza Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *NDSS*, 2020.

[252] Sergej Schumilo, Cornelius Aschermann, Ali Reza Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

[253] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX security symposium (USENIX Security 17)*, 2017.

[254] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1681–1692. Association for Computing Machinery, 2022.

[255] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.

[256] Yasser Shalabi, Mengjia Yan, Nima Honarmand, Ruby B. Lee, and Josep Torrellas. Record-Replay Architecture as a General Security Framework. *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 180–193, 2018.

[257] Md Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. XI Commandments of Kubernetes Security: A Systematization of Knowledge related to Kubernetes Security Practices. *IEEE Secure Development (SecDev)*, pages 58–64, 2020.

[258] Shazibul Islam Shamim. Mitigating security attacks in Kubernetes manifests for security best practices violation. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1689–1690, 2021.

[259] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. Tardis: Coverage-Guided Embedded Operating System Fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[260] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds. In *USENIX Security Symposium*, 2022.

[261] Hossein Siadati, Sima Jafarikhah, Elif Sahin, Terrence Hernandez, Elijah Tripp, Denis Khryashchev, and Amin Kharraz. DevPhish: Exploring Social Engineering in Software Supply Chain Attacks on Developers. In *2024 IEEE 15th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 517–523. IEEE, 2024.

[262] Richard E Smith. A contemporary look at Saltzer and Schroeder's 1975 design principles. *IEEE Security & Privacy*, 10(6):20–25, 2012.

[263] Snyk Ltd. Snyk cli documentation, 2025. Accessed: 2025-04-12.

[264] Sonatype. Nancy. `https://pkg.go.dev/github.com/sonatype-nexus-community/nancy`, 2024. Online; accessed 20 June 2024; Language version go1.22.

[265] Sonatype. OSS Index. `https://ossindex.sonatype.org/`, 2024. Online; accessed 20 June 2024;.

[266] Sonatype. PyPI crypto-stealer targets Windows users, revives malware campaign. `https://www.sonatype.com/blog/pypi-crypto-stealer-targets-windows-users-revives-malware-campaign`, May 2024. Online; accessed 9 June 2024.

[267] Sonatype. State of the Software Supply Chain. `https://www.sonatype.com/state-of-the-software-supply-chain/2024/`, 2024. Online; accessed 26 October 2025;.

[268] Alessandro Sorniotti, Michael Weissbacher, and Anil Kurmus. Go or no go: Differential fuzzing of native and c libraries. In *2023 IEEE Security and Privacy Workshops (SPW)*, pages 349–363. IEEE, 2023.

[269] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: Automated IoT firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, 2019.

[270] StackRox. KubeLinter. `https://kubelinter.io`, 2024.

[271] Akihiro Suda. Don't use net=host. `https://medium.com/nttlabs/dont-use-host-network-namespace-f548aeeef575`, 2020.

[272] Tetragon. ebpf-based security observability and runtime enforcement, 2025. Accessed: 2025-04-12.

[273] The Go Programming Language. generate package - Go Packages. `https://pkg.go.dev/cmd/go/internal/generate`, 2024. Online; accessed 29 May 2024.

[274] The Go Programming Language. Go doc - effective go. `https://go.dev/doc/effective_go`, 2024. Online; accessed 2 July 2024.

[275] The Go Programming Language. Go Module Mirror. `https://proxy.golang.org/`, 2024. Online; accessed 27 May 2024.

[276] The Go Programming Language. Go Modules Reference. https://go.dev/ref/mod, 2024. Online; accessed 24 May 2024.

[277] The Go Programming Language. The Go Programming Language Specification. https://go.dev/ref, 2024. Online; accessed 10 June 2024; Language version go1.22.

[278] The Go Programming Language. Package initialization. https://go.dev/ref/spec#Package_initialization, 2024. Online; accessed 27 May 2024; Language version go1.22.

[279] The Go Programming Language. Package names. https://go.dev/blog/package-names, 2024. Online; accessed 28 May 2024.

[280] The Go Programming Language. A Quick Guide to Go's Assembler, 2024.

[281] The Go Programming Language. The Go Compiler. https://go.dev/src/cmd/compile/README, 2024. Online; accessed 27 May 2024.

[282] The Go Programming Language. The Laws of Reflection. https://go.dev/blog/laws-of-reflection, 2024. Online; accessed 28 May 2024.

[283] The Linux Foundation. Clock sources, Clock events, sched_clock() and delay timers.

[284] The Linux Kernel Archives. Using the Linux Kernel Tracepoints. https://docs.kernel.org/trace/tracepoints.html, 2025.

[285] The Linux Kernel Community. seccomp(2) — linux manual page. https://man7.org/linux/man-pages/man2/seccomp.2.html, 2024. Accessed: 2025-04-11.

[286] The Linux Kernel Community. x86_64 system call table (syscall_64.tbl), 2025. Accessed: 2025-04-12.

[287] Nektarios Georgios Tsoutsos and Michail Maniatakos. Anatomy of Memory Corruption Attacks and Mitigations in Embedded Systems. *IEEE Embedded Systems Letters*, 10(3):95–98, 2018.

[288] UK National Cyber Security Centre (NCSC). Secure design principles: Guides for the design of cyber secure systems. Technical report, NCSC, 2020.

[289] US Cybersecurity and Infrastructure Security Agency. Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software. Technical report, CISA, 2023.

[290] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. Preventing Dynamic Library Compromise on Node.js via RWX-Based Privilege Reduction. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1821–1838, 2021.

[291] Wenya Wang, Xingwei Lin, Jingyi Wang, Wang Gao, Dawu Gu, Wei Lv, and Jiashui Wang. HODOR: Shrinking Attack Surface on Node.js via System Call Limitation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2800–2814, 2023.

[292] Song Wei, Kun Zhang, and Bibo Tu. Hyperbench: A benchmark suite for virtualization capabilities. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–22, 2019.

[293] Alex Williams. Giving Go a Go: Simplifying Cloud Infrastructure Development. `https://cacm.acm.org/blogcacm/giving-go-a-go-simplifying-cloud-infrastructure-development/`, 2024.

[294] Yongzheng Wu, Sai Sathyanarayan, Roland HC Yap, and Zhenkai Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *Computer Security–ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings 17*, pages 859–876. Springer, 2012.

[295] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. Wolf at the door: Preventing install-time attacks in npm with latch. In *ACM ASIA Conference on Computer and Communications Security (ASIACCS)*, pages 1139–1153, 2022.

[296] Xen project. Hvmloader.

[297] Qingxin Xu, Yu Gao, and Jun Wei. An Empirical Study on Kubernetes Operator Bugs. In *33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1746–1758, 2024.

[298] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 9–16. IEEE, 2012.

[299] Qiuchen Yan and Stephen McCamant. Fast PokeEMU: Scaling Generated Instruction Tests Using Aggregation and State Chaining. *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2018.

[300] Nanzi Yang, Wenbo Shen, Jinku Li, Xunqi Liu, Xin Guo, and Jianfeng Ma. Take over the whole cluster: Attacking Kubernetes via excessive permissions of third-party applications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3048–3062, 2023.

[301] Xia Yang, Jian Lei, and Guang-ze Xiong. Inter-partition Information Flow Control for High-Assurance Embedded Systems. In *2009 WRI World Congress on Computer Science and Information Engineering*. IEEE, 2009.

[302] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019.

[303] Ahmed Zerouali, Ruben Opdebeeck, and Coen De Roover. Helm charts for Kubernetes applications: Evolution, outdatedness and security risks. In *IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 523–533, 2023.

[304] Zhaoqi Zhang, Panpan Qi, and Wei Wang. Dynamic malware analysis with feature engineering and feature learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 1210–1217, 2020.

[305] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[306] Yaowen Zheng, Yuekang Li, Cen Zhang, Hongsong Zhu, Yang Liu, and Limin Sun. Efficient greybox fuzzing of applications in Linux-based IoT devices via enhanced user-mode emulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[307] Hui Zhu and Christian Gehrmann. Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine. In *14th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 129–137, 2022.

[308] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

Author's publications

The following previously published material has been, in parts verbatim, included in this thesis.

1. Cesarano, C., Cinque, M., Cotroneo, D., De Simone, L., & Farina, G. (2023, June). *IRIS: a Record and Replay Framework to Enable Hardware-assisted Virtualization Fuzzing.* In 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (pp. 389-401). IEEE.
2. Cesarano, C., Andersson, V., Natella, R., & Monperrus, M. (2023, November). *GoSurf: Identifying Software Supply Chain Attack Vectors in Go.* In Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (pp. 33-42).
3. Cesarano, C., & Natella, R. (2025). *KubeFence: Security Hardening of the Kubernetes Attack Surface.* In 2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).
4. Cesarano, C., Monperrus, M., & Natella, R. (2025). *GoLeash: Mitigating Golang Software Supply Chain Attacks with Runtime Policy Enforcement.* arXiv preprint arXiv:2505.11016.
5. Cesarano, C., & Natella, R. (2025). *FuzzBox: Blending Fuzzing into Emulation for Binary-Only Embedded Targets.* arXiv preprint arXiv:2509.05643. [ACCEPTED to Springer Cybersecurity Journal]

The following publications are related to the different aspects covered in this thesis but have not been included.

1. Cesarano, C., Cotroneo, D., & De Simone, L. (2022, October). *Towards assessing isolation properties in partitioning hypervisors.* In 2022 IEEE

international symposium on software reliability engineering workshops (ISSREW) (pp. 193-200). IEEE.

2. Cesarano, C. (2023, October). *Security Assessment and Hardening of Fog Computing Systems*. In 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW) (pp. 22-25). IEEE.

3. Cesarano, C., & Natella, R. (2024, April). *Securing an Application Layer Gateway: An Industrial Case Study*. In 2024 19th European Dependable Computing Conference (EDCC) (pp. 75-80). IEEE.

4. Cesarano, C., Foggia, A., Roscigno, G., Andreani, L., & Natella, R. (2025). *GENIO: Synergizing Edge Computing with Optical Network Infrastructures*. IEEE Communications Magazine.

5. Cesarano, C., Foggia, A., Roscigno, G., Andreani, L., & Natella, R. (2025). *Security-by-Design at the Telco Edge with OSS: Challenges and Lessons Learned*. In 2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S).