# Crossing Shifted Moats: Replacing Old Bridges with New Tunnels to Confidential Containers

Enriquillo Valdez
IBM Research
Yorktown Heights, USA
rvaldez@us.ibm.com

Salman Ahmed
IBM Research
Yorktown Heights, USA
sahmed@us.ibm.com

Zhongshu Gu
IBM Research
Yorktown Heights, USA
zgu@us.ibm.com

Christophe de Dinechin
Red Hat
Valbonne, France
cdupontd@redhat.com

Pau-Chen Cheng
IBM Research
Yorktown Heights, USA
pau@us.ibm.com

Hani Jamjoom
IBM Research
Yorktown Heights, USA
jamjoom@us.ibm.com

## ABSTRACT

The Confidential Containers (CoCo) project, as an open-source community initiative, inherits the system architecture of Kata Containers while integrating confidential computing to protect cloud-native container workloads. However, there exists a misalignment in the threat model and trusted computing base (TCB) between Kata Containers and confidential computing. The shifted trust boundaries could potentially expose a range of vulnerabilities, particularly in scenarios where a malicious actor on the host gains access to the CoCo's unprotected control interface. This paper conducts a thorough examination of CoCo's system architecture, exploring the attack surface resulting from the discord in trust boundaries. We have assessed all API endpoints of CoCo's control interface, categorizing them based on their security properties. Drawing from these insights, we have developed a bifurcation approach to splitting CoCo's control interface. This involves establishing an owner-side controller and minimizing the capabilities of the existing host-side controller. Under this framework, the host-side controller is exclusively responsible for allocating and recycling compute resources, while dedicated workload owners can directly manage their containers through alternative secure tunnels. This approach ensures seamless integration with cloud-native orchestration layers and aligns CoCo with the threat model of confidential computing. By doing so, it effectively prevents untrusted hosts from accessing confidential data and interfering with the execution of workloads within protected domains.

## CCS CONCEPTS

• **Security and privacy → Systems security**.

## KEYWORDS

Confidential Computing; Confidential Containers

*CCS '24, October 14–18, 2024, Salt Lake City, UT, USA*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0636-3/24/10.
https://doi.org/10.1145/3658644.3670352

**ACM Reference Format:**
Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Christophe de Dinechin, Pau-Chen Cheng, and Hani Jamjoom. 2024. Crossing Shifted Moats: Replacing Old Bridges with New Tunnels to Confidential Containers. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670352

## 1 INTRODUCTION

Confidential computing has fundamentally changed the threat landscape for computation on third-party machines. It promotes the concept of a minimized trusted computing base (TCB), wherein the root-of-trust comprises the underlying processors and extends to a designated set of CPU-attested software modules. Notably, the host software stack is excluded from this TCB. Adhering to this small TCB shifts the responsibility of attesting the platform to the workload owners.

At the same time, the cloud industry is embracing a cloud-native model, characterized by packaging applications as containers and relying on orchestration tools, such as `Kubernetes` or `OpenShift`, for managing, scaling, and load-balancing containerized services.

Confidential Containers (CoCo) [10] project is a Cloud Native Computing Foundation (CNCF) open-source initiative that aims to bridge these two endeavors by protecting cloud-native container deployments with confidential computing. CoCo inherits the execution model and system architecture of the Kata Containers [12] project, which encapsulates containers with lightweight virtual machines (VMs). CoCo enhances the security by substituting regular VMs in Kata with confidential VMs and introducing additional service components, including key broker/management, image building/registry, and attestation, to facilitate the container deployment.

At its core, the trust boundaries of confidential computing and Kata Containers exhibit an inherent misalignment. Confidential computing provides adversaries with an advantage in controlling the host software stack, which is no longer deemed trustworthy. In contrast, the Kata model retains a feature-rich workload controller that operates on the host. This controller receives commands from the orchestration layer, such as the `Kubernetes` control plane, and governs the in-VM `kata-agent` responsible for managing container workloads. Adhering to the original design of Kata introduces a series of new security vulnerabilities. These vulnerabilities, in turn, heighten the risks of information leaks and execution tampering within protected domains, mitigating the security benefits brought by confidential computing.

In this paper, we undertake a systematic assessment of the attack surface within the current implementation of CoCo. We investigate how adversaries in the current setting could access confidential data or manipulate the execution of container workloads within confidential VMs. We have conducted a series of security experiments that involve the misuse of individual or combined CoCo API endpoints. Our experiments reveal that, by exploiting CoCo's unprotected control interface, adversaries can achieve a spectrum of attack effects. These effects include leaking a neighboring container's memory, obtaining runtime metrics, controlling the execution of target containers, disrupting critical system services by manipulating the date and time settings, and overwriting critical files within the confidential VMs. Notably, these effects fundamentally violate the security principles of confidential computing, which is designed to prevent such adversarial actions. We identified the misalignment issue through our involvement in the research and development of CoCo. Our findings are not limited to CoCo. They extend to any software embarking on the integration of confidential computing into its execution flow. Our paper aims to raise awareness about the need to scrutinize existing designs under the new threat model of confidential computing and re-examine the control interface and data crossing the shifted trust boundary.

To bridge this security gap, we have developed a bifurcation solution to split CoCo's control interface. This design partitions all API endpoints of CoCo into two distinct controllers, based on their inherent security properties. The *host-side controller* is now limited in its scope, primarily responsible for resource allocation during the initialization phase and resource recycling in the termination phase. It no longer holds the privilege to access private data or assert control over operations within confidential VMs. In contrast, authorized workload owners gain the ability to directly manage their containers through an *owner-side controller*. This controller utilizes CoCo's attestation infrastructure to establish a secure tunnel directly with the `kata-agent` managing the workload within the confidential VM. This redesign serves a dual purpose: it aligns CoCo with the threat model of confidential computing, while ensuring seamless integration with cloud-native orchestration layers.

**Responsible Disclosure.** Given that CoCo is an open-source project still in the developmental phase, vulnerability reports are managed through public GitHub issues [2]. Our findings have been promptly reported to the CoCo community, and we actively engaged in discussions about these issues during community meetings. We have submitted two *pull requests* [1] [2] that include the implementation of the split control interface. This collaborative and transparent approach ensures that identified vulnerabilities are addressed responsibly within the open-source community.

**Roadmap.** The rest of this paper is structured as follows. Section 2 provides the background knowledge of Kata Containers and confidential computing. We explain how CoCo integrates both technologies. Section 3 delves into the vulnerable control interface of CoCo stemming from the misaligned threat model. We categorize potential attacks into two primary groups, *information leakage* (IL) and *execution tampering* (ET), and investigate their security implications respectively. Section 4 presents five concrete security experiments, demonstrating what can be achieved by attackers to exploit CoCo's control interface. Section 5 describes our proposed defense, which involves a bifurcation of CoCo's control interface to serve two distinct controllers. Section 6 evaluates our defense approach from both security and performance perspectives. Section 7 discusses the compatibility of these changes with the broader container ecosystem and highlights potential avenues for future research. Section 8 provides insights into related works in the field. Section 9 concludes the paper.

[1]https://github.com/kata-containers/kata-containers/pull/9159
[2]https://github.com/kata-containers/kata-containers/pull/9752

## 2 BACKGROUND

To comprehend the execution flow and system architecture of CoCo, we begin by providing background knowledge about Kata Containers and confidential computing. This understanding sets the stage for a detailed explanation of how CoCo integrates both technologies and the security implications of such integration.

### 2.1 Containers → Kata Containers

Containers are self-contained and fully functional executable packages. They encompass not only the application code but also the necessary runtime components, libraries, and system configurations, enabling them to operate consistently across different computing environments. The isolation and resource management of containers rely on the underlying security features of the Linux kernel. These features include `namespaces`, `cgroups`, `capabilities`, `SELinux`, `AppArmor`, `seccomp`, among others. Despite these security measures in place, running containers directly on bare-metal machines is considered insecure [15–17, 51], potentially jeopardizing the underlying host system.

The fundamental concept behind Kata Containers revolves around the enforcement of isolation by encapsulating container workloads within VMs. This approach is designed to contain the impact of potential attacks within the VM boundaries, preventing them from cascading into the host system. On the other side, Kata Containers strives to achieve performance comparable to bare-metal containers, by using lightweight MicroVMs to expedite boot time and minimize resource utilization.

We provide an overview of the workflow for creating and managing containers using Kata Containers. The process begins with a single instance of `containerd-shim-kata-v2` (abbreviated as `kata-shim`), which is responsible for receiving API calls from the container runtime engine, e.g., `containerd`. When tasked with creating a container, `kata-shim` initiates the process by invoking a hypervisor to start a VM. Within this VM, a long-running process known as the `kata-agent` is launched during boot time. The `kata-agent` assumes the role of a supervisor, responsible for creating and managing pod/containers within the VM. The term `sandbox` is frequently used in the `kata-agent`'s API endpoints, and it typically refers to the execution environment within the VM. Subsequently, `kata-shim` (running outside the VM) communicates with the `kata-agent` (running within the VM) using the `gRPC` APIs via a `vsock` channel. The `containerd` pulls container images from an image registry to the host machine. The `kata-shim` then mounts these container images within the VM and instructs the `kata-agent` to

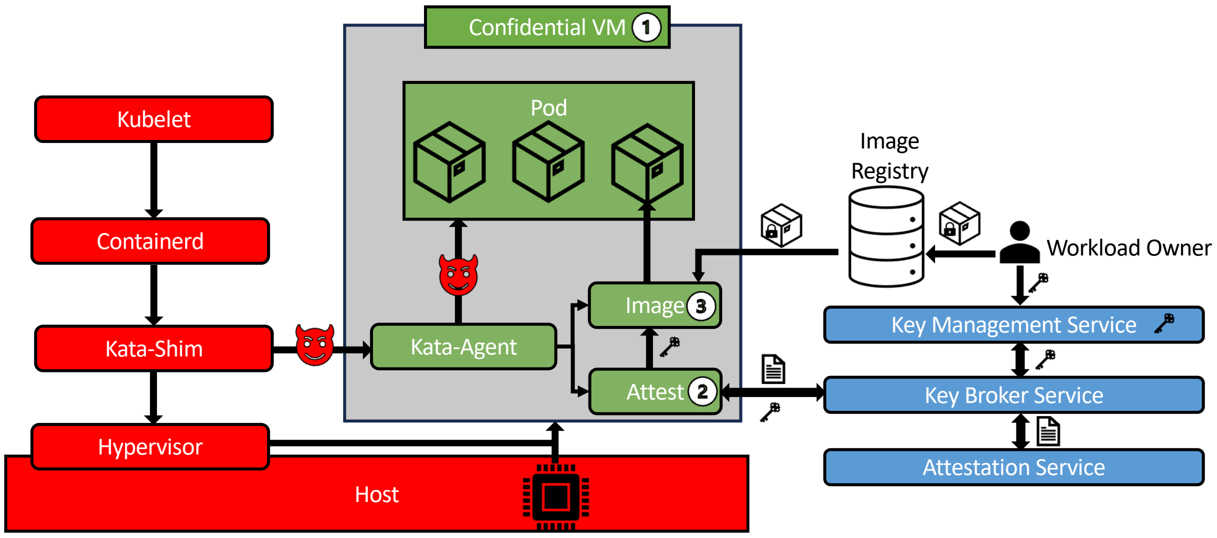


**Figure 1: System Architecture and Workflow of Confidential Containers**

create and start the containers. The entire life cycle of the pod and its constituent containers is managed via the control interface between the `kata-shim` and the `kata-agent`.

**Threat Model of Kata Containers.** It is important to emphasize that Kata is specifically designed to provide defense against potentially malicious container workloads. As such, it operates under the threat model that the host software components, *e.g.,* `containerd`, `kata-shim`, the hypervisor, and the communication between `kata-shim` and `kata-agent`, are trusted.

## 2.2 Confidential Computing

Confidential computing leverages hardware-based Trusted Execution Environments (TEEs) to protect computation within protected domains on third-party machines. The primary focus of these technologies centers on protecting data-in-use, which pertains to data loaded into main memory, while also minimizing the root-of-trust to the processors. Despite variations in implementation and terminology across these technologies, they adhere to fundamental security principles that align with similar system designs. These principles encompass several key aspects: (1) introduction of new execution modes or privilege levels for workloads and workload monitors, (2) enclosing workload management functions to vendor-signed firmware, (3) secure or measured launch of trusted components, (4) enforcing memory access controls or providing memory encryption protection, and (5) allowing remote workload owners to verify the integrity of the workloads and the authenticity of the underlying processors.

The granularity of protected domains can vary depending on the specific TEEs employed. For instance, Intel Software Guard Extensions (SGX) [37] offers protection at the process level, protecting a portion of memory referred to as an `enclave` within a single process. In recent times, several CPU vendors have introduced VM-based TEEs that integrate with hardware-assisted virtualization, providing execution protection at the VM level. Representative systems in this category include AMD Secure Encrypted Virtualization (SEV) [27, 28, 42], Intel Trust Domain Extensions (TDX) [26], IBM Secure Execution (SE) [23], Protected Execution Facility (PEF) [22], and ARM Confidential Compute Architecture (CCA) [32]. VM-based confidential computing benefits from a well-defined trust boundary established through virtualization isolation. This approach enables the execution of unmodified applications and containers within VMs without the need of refactoring application code.

**Threat Model of Confidential Computing.** Confidential computing has introduced a new threat model. This model operates under the assumption that potential adversaries may have physical or remote access to a machine and could potentially gain control of the entire host software stack, including the boot firmware, host operating system (OS), and hypervisor. The primary objective of this model is to preserve the confidentiality and integrity of the code and data contained within the protected domains. However, it is important to note that these techniques cannot guarantee availability. Adversaries with control over the host software can manipulate compute resources and potentially launch Denial of Service (DoS) attacks.

## 2.3 CoCo = Kata + Confidential Computing

CoCo represents a convergence [1] of these two key technologies: Kata Containers and VM-based confidential computing. It ensures that the deployed container workloads receive runtime protection in terms of both confidentiality and integrity. Furthermore, to adapt to confidential computing, CoCo also introduces a suite of supporting services, including attestation, key provisioning, and encrypted container images.

The overall system architecture and workflow of CoCo closely resemble that of Kata Containers, with some adaptations made for confidential computing. In this context, we highlight three major differences (numbered in Figure 1) introduced by CoCo:

① **Confidential VM.** In CoCo, the `kata-shim` launches a confidential VM rather than a regular VM. This confidential VM initially contains no secret and boots with a modified `kata-agent`. As of the time of writing, CoCo can support three VM-based TEEs: AMD SEV, Intel TDX, and IBM SE.

② **Attestation and Key Provisioning.** Unlike Kata, where the launched VM is inherently trusted due to its threat model, the confidential VM in CoCo must prove its trustworthiness through attestation [14]. To achieve this, CoCo's `kata-agent` spawns an `attester` to retrieve signed evidence from the underlying TEE. This evidence includes measurements of both the hardware platform and guest software stack. The `attester` then communicates with the Key Broker Service (KBS) [48], which in turn verifies the evidence with the attestation service. Once verified, the KBS obtains the keys from the Key Management Service (KMS) and delivers them securely to the `attester`. The keys are used for unpacking container images, *i.e.,* decrypting image files and verifying the digital signatures.

③ **Image Pulling and Unpacking.** Unlike Kata, `containerd` and `kata-shim` should no longer control the image pulling as they have been excluded from the TCB in confidential computing. Therefore, in CoCo, the workload owner is responsible for signing and encrypting the container images, and pushing these encrypted images to the image registry during the preparation stage. Subsequently, the `kata-agent` requests the image management library to pull these encrypted container images and unpack them using the keys obtained from the KBS.

**Threat Model of Confidential Containers.** It is evident that confidential computing operates under a stronger threat model and a smaller TCB in comparison to Kata Containers. To align with the security principles of confidential computing, CoCo must adopt an equivalent threat model consistent with confidential computing standards. This adjustment ensures that CoCo maintains the heightened security posture necessary for protecting container workloads within confidential VMs.

## 3 SECURITY IMPLICATIONS OF SHIFTED TRUST BOUNDARIES

As we have mentioned in Section 2, to establish a secure execution environment for container deployment, CoCo has added several new hardening features. However, despite these enhancements, the host control plane continues to manage container workloads via the control interface between `kata-shim` and `kata-agent` at runtime. It is essential to note that this control interface lacks adequate protection and still relies on gRPC APIs through a `vsock` channel, mirroring the setup in Kata Containers.

In the CoCo's threat model, `kata-shim` and `kata-agent` have been placed into different security realms: `kata-shim` is no longer considered trusted and `kata-agent` remains protected in the confidential VM. This control interface represents a substantial attack surface, exposing the code and data within confidential VMs to exploitation if a malicious actor gains control over the `kata-shim`.

The CoCo community has made some initial attempts to tackle this issue. One such effort involves enabling workload owners to create a permission list that can disable specific sensitive APIs, like `ExecProcess` and `ReseedRandomDev` [38], to guard against potential attacks from the host software stack. Microsoft Azure further allows the workload owner to define a comprehensive security policy [3] for `kata-agent`'s API calls and their associated parameters. The integrity of the policy can be verified by the `kata-agent` and the enforcement of the policy uses the Open Policy Agent (OPA).

However, it is important to note that the existing API endpoints serve a broad spectrum of functions related to container management. Disabling certain APIs cannot provide a solution for mitigating all types of attacks, as we illustrate in Section 4. Furthermore, such actions may also have a substantial impact on the operational efficiency and functionality available to workload owners.

As the control interface remains unprotected, adversaries (*e.g.,* rogue host administrators) can intercept incoming API commands from workload owners, modify these commands, and even inject new commands into the communication channel. This capability empowers them to achieve two primary objectives: (1) extracting sensitive information from the workloads and (2) manipulating the execution of workloads, within confidential VMs. Following a comprehensive analysis of 38 API endpoints of `kata-agent`, as detailed in Table 1, we highlight the API endpoints that can potentially serve as attack vectors for enabling *information leakage* and *execution tampering* attacks.

### 3.1 Information Leakage

We have identified nine API endpoints (marked with IL in Table 1) that, either individually or in combination, can potentially serve as attack vectors for leaking sensitive information from workloads within confidential VMs. These API endpoints include `CreateContainer`, `ExecProcess`, `GetGuestDetails`, `GetMetrics`, `PullImage`, `ReadStderr`, `ReadStdout`, `StartContainer`, `StatsContainer`. In Section 4.1, we demonstrate how adversaries can access the memory of a victim's container by deploying a rogue container alongside the victim's container. Additionally, in Section 4.2, we illustrate how adversaries can retrieve the runtime metrics of workloads within the confidential VM. These attacks breach the confidentiality protection enforced by TEEs.

### 3.2 Execution Tampering

We have identified thirteen API endpoints (marked with ET in Table 1) that have the potential to serve as attack vectors for execution tampering attacks. Attackers can exploit such API endpoints to disrupt the execution process. This disruption could have severe consequences, particularly if a critical service is operating within confidential VMs. A few examples of these API endpoints include `RemoveContainer`, `ExecProcess`, `PauseContainer`, `ResumeContainer`, `CopyFile`, `ReseedRandomDev`, etc. In Section 4.3, we illustrate how adversaries can cause execution tampering by pausing, resuming, or even removing a confidential container. In Section 4.4, we demonstrate an attack where adversaries can disrupt critical services and functionalities by manipulating the date and time settings of a confidential VM. Furthermore, in Section 4.5, we show that adversaries can add or overwrite any critical files in the

**Table 1: This is a list of `Kata-agent` API endpoints, showing the attack types, partitioning labels, and description of functionalities. The labels `Host-Exclusive` and `Owner-Exclusive` indicate that an API is exclusively assigned to either the host-side or owner-side control plane. The labels `Sanitized`, `Shared`, and `Switch` denote the extent of API sharing between the control planes. `Sanitized` means that the host-side invocation of an API undergoes sanitization to prevent *information leakage* (IL) or *execution tampering* (ET). `Shared` denotes that both control planes have access to the same API functionality. `Switch` indicates the host-side access to the API is restricted to the sandbox initialization phase.**

| API endpoint | Attack Types | Partitioning Label | Description |
|---|---|---|---|
| `AddARPNeighbors` | | Host-Exclusive | Adds ARP neighbors |
| `CreateSandbox` | | Host-Exclusive | Creates a sandbox environment |
| `DestroySandbox` | | Host-Exclusive | Destroys a sandbox environment |
| `GetIPTables` | | Host-Exclusive | Retrieves iptables of a sandbox |
| `GetVolumeStats` | | Host-Exclusive | Gets volume status of a guest |
| `ResizeVolume` | | Host-Exclusive | Resizes a volume of a guest |
| `SetIPTables` | | Host-Exclusive | Sets iptables of a sandbox |
| `UpdateInterface` | | Host-Exclusive | Updates an interface in a sandbox |
| `UpdateRoutes` | | Host-Exclusive | Updates routes in a sandbox |
| `CopyFile` | ET | Owner-Exclusive | Writes files into the VM's `/run` directory |
| `ExecProcess` | ET, IL | Owner-Exclusive | Creates and runs a process in a container |
| `PauseContainer` | ET | Owner-Exclusive | Pauses execution of a container |
| `PullImage` | ET, IL | Owner-Exclusive | Pulls an image in a sandbox |
| `RemoveContainer` | ET | Owner-Exclusive | Removes a container from a sandbox |
| `ReseedRandomDev` | ET | Owner-Exclusive | Sets seed on a random device |
| `ResumeContainer` | ET | Owner-Exclusive | Resumes a paused container |
| `SetGuestDateTime` | ET | Owner-Exclusive | Sets time of a sandbox |
| `StatsContainer` | IL | Owner-Exclusive | Gets cgroup and network statistics |
| `TtyWinResize` | | Owner-Exclusive | Resizes tty rows and columns |
| `UpdateContainer` | ET | Owner-Exclusive | Updates a container's resources |
| `Check` | | Shared | Returns status of kata-agent |
| `GetOOMEvent` | | Shared | Retrieves out of memory (OOM) event |
| `ListInterfaces` | | Shared | Lists interfaces |
| `ListRoutes` | | Shared | Lists routes |
| `OnlineCPUMem` | | Shared | Reports online CPUs |
| `Version` | | Shared | Retrieves kata-agent and API version |
| `WaitProcess` | | Shared | Waits for a container's signal |
| `CloseStdin` | | Sanitized | Closes a container's Stdin |
| `GetMetrics` | IL | Sanitized | Gets resource usage of a sandbox |
| `GetGuestDetails` | IL | Sanitized | Gets information of a sandbox and its agent |
| `ReadStderr` | IL | Sanitized | Reads a container's Stderr |
| `ReadStdout` | IL | Sanitized | Reads a container's Stdout |
| `SignalProcess` | ET | Sanitized | Sends signal to a container |
| `WriteStdin` | ET | Sanitized | Writes to a container's Stdin |
| `CreateContainer` | ET, IL | Switch | Creates a container |
| `StartContainer` | ET, IL | Switch | Starts execution of a container |
| `AddSwap` | | Not Supported | Adds a swap file to a sandbox |
| `MemHotplugByProbe` | | Not supported | Notifies a hotplug memory event |

sandbox or container configurations as well as the `rootfs`. These attacks breach the integrity protection enforced by TEEs.

## 4 SECURITY EXPERIMENTS

We have performed five distinct security experiments to demonstrate how the unprotected CoCo control interface can be exploited. In the security experiments, we assume the role of an adversary who has successfully infiltrated the host system and gained control over `kata-shim`. With access to the host file system, the adversary can obtain the `sandbox` and `container` IDs, which are used for identifying target containers within confidential VMs. Leveraging these acquired IDs, the adversary sends `gRPC` API commands to the `kata-agent` through the `vsock` channel to carry out the attacks. To facilitate the process of invoking APIs, we rely on the `kata-agent-ctl` [47], a utility tool provided by Kata Containers, to exercise the control interface of `kata-agent`. This tool allows us to directly issue API commands and receive results from the `kata-agent`, in the same position as the `kata-shim`.

**Hardware/Software Configuration for Experiments.** Our test machine has a two-socket configuration with 4th Generation Intel Xeon Scalable (Sapphire Rapids) processors (96 cores running at 1.5GHz) and 30 GB of memory. The host OS is Linux kernel 5.15-SPR.BKC.PC.v8.8. We utilize Intel TDX (version 1.0) as the TEE to initiate the confidential VMs in CoCo. It is worth noting that our experiments are not tied to any specific TEEs. The results should be broadly applicable to other TEEs that are supported in CoCo.

### 4.1 Leaking Neighbor Container's Memory

This attack extracts the memory pages from one or more processes within a confidential container to retrieve sensitive data such as application code, credentials, and encryption keys. The private memory pages in a confidential VM should only contain ciphertext or potentially all-zero [24]. Thus, any attempts by an adversary to read the memory pages from outside the VM should only get encrypted or all-zero content. However, we show that an attacker can bypass the memory encryption protection by exploiting the unprotected control interface between the `kata-shim` and `kata-agent` to perform memory dumping within the VM. As a result, all the dumped memory is in plaintext and is ready for memory forensic analysis.

The key enabler of this attack is that one container can observe another container's processes if they are in the same pod of a confidential VM. This mutual process visibility opens the door to potential misuse. A rogue administrator with access to the host control plane can exploit this capability to access and read the memory of another process.

To illustrate this attack, we assume that a container (referred to as the *victim*) is already operational within a confidential VM as the attack unfolds. Then the attack proceeds step by step as follows:

- Initially, the adversary examines the host file system to extract the `sandbox` ID. The `sandbox` ID is located in the host directory `/run/kata-containers/shared/sandboxes/` for the target confidential VM.
- Following this, the adversary creates a new container within the same `sandbox`, utilizing the previously obtained `sandbox` ID and the `kata-agent-ctl` tool. This newly created container is dubbed the *attacker* container and operates as a neighbor to the *victim* container. The *attacker* container has the necessary code or scripts to execute the subsequent phases of the attack.
- Once the *attacker* container becomes operational within the `sandbox`, the embedded scripts inspect the memory mappings of all processes running within the *victim* container, dump them, and transfer them to the host file system for offline analysis for sensitive content.
- Finally, the *attacker* container terminates after completing its memory-dumping operation.

**Table 2: Statistics of dumping memory from neighbor processes by the *attacker* container**

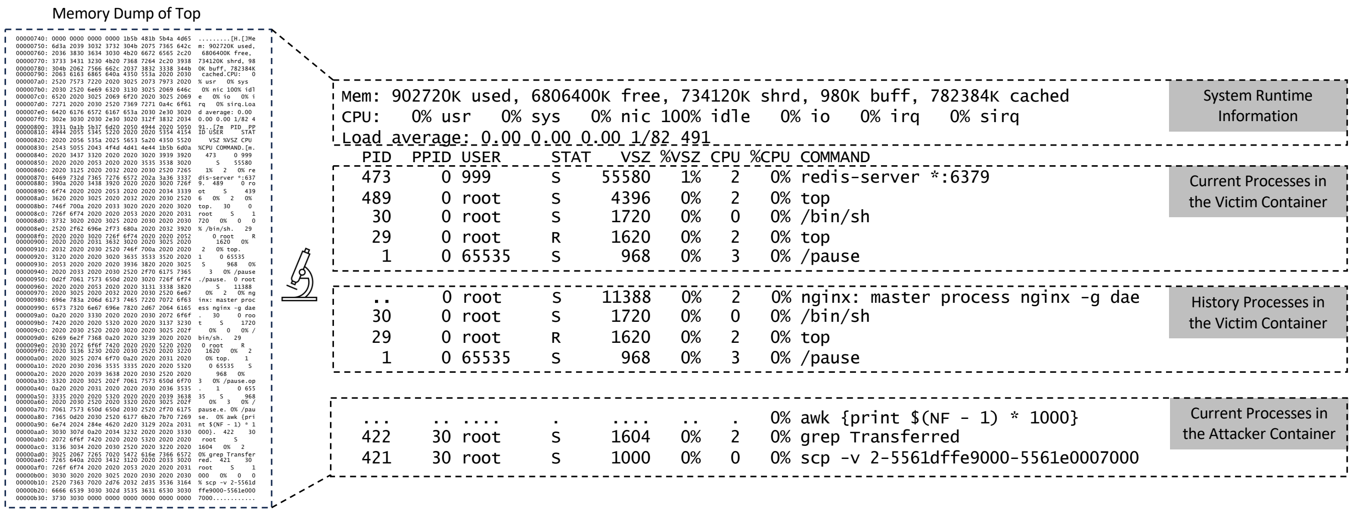
| Victim Container | `nginx` | `alpine-top` | `alpine-custom` |
|---|---|---|---|
| Raw Memory | 1728 KB | 172 KB | 156 KB |
| Non-zero Memory | 43.57 KB | 1.18 KB | 0.51 KB |
| Time to Dump | 0.78 s | 0.34 s | 0.24 s |
| Time to Transfer to Host | 0.3 s | 0.2 s | 0.2 s |

We performed this attack on three *victim* containers (`nginx`, `alpine-top`, and `alpine-custom`). The `nginx` container initiates an `nginx` web server upon startup. The `alpine-top` executes the `top` command within an `alpine` Linux environment at container launch. Additionally, we created a customized container named `alpine-custom`, which starts a program containing an embedded secret. Our objective is to search for this secret within the memory dump as evidence, determining the success of the attack.

We also prepared an `alpine`-based *attacker* container. We first launched all the *victim* containers in a confidential VM using the CoCo stacks (i.e., `containerd`, `kata-shim`, `kata-agent`, KBS, etc.). Once the *victim* containers were operational, we started the *attacker* container by invoking the `PullImage`, `CreateContainer`, and `StartContainer` via the `kata-agent-ctl` tool. The *attacker* container performed the memory dumping of the processes of `nginx` (master process), `top`, and the customized program, then transferred the dumped memory to the host. Table 2 presents the statistics regarding memory dumping. The data indicate that all attacks can be completed approximately within one second, thereby making detection challenging.

During the offline analysis, we scrutinized the dumped raw memory content to identify sensitive and secret information. In this experiment, we assumed the role of the adversary and employed a straightforward byte-by-byte reading method to locate the secret content. For instance, as illustrated in Figure 2, within the memory dump of the `top` process, we could extract information related to all running processes within the system. Additionally, we validated the adversary's ability to identify sensitive data from the memory dump by possessing prior knowledge of some secret data that the custom victim container writes to its heap. Our simple analysis successfully extracted the secret data from the memory dump of the process of the custom container. However, it is important to note that in reality, the offline analysis process typically requires more advanced and comprehensive memory forensics approaches [39–41].

This experiment demonstrates the possibility of exploiting CoCo APIs to inject an attack container into a confidential VM, leading to


Memory Dump of Top
System Runtime Information
Current Processes in the Victim Container
History Processes in the Victim Container
Current Processes in the Attacker Container

```
00000740: 0000 0000 0000 0000 1b5b 481b 5b4a 4d65  .........[H.[JMe
00000750: 6d3a 2039 3032 3732 304b 2075 7365 642c  m: 902720K used,
00000760: 2036 3830 3634 3030 4b20 6672 6565 2c20   6806400K free,
00000770: 3733 3431 3230 4b20 7368 7264 2c20 3938  734120K shrd, 98
00000780: 304b 2062 7566 662c 2037 3832 3338 344b  0K buff, 782384K
00000790: 2063 6163 6865 640a 4350 553a 2020 2030   cached.CPU:   0
000007a0: 2520 7573 7220 2020 3025 2073 7973 2020  % usr   0% sys
000007b0: 2030 2520 6e69 6320 3130 3025 2069 646c   0% nic 100% idl
000007c0: 6520 2020 3025 2069 6f20 2020 3025 2069  e   0% io   0% i
000007d0: 7271 2020 2030 2520 7369 7271 0a4c 6f61  rq   0% sirq.Loa
000007e0: 6420 6176 6572 6167 653a 2030 2e30 3020  d average: 0.00
000007f0: 302e 3030 2030 2e30 3020 312f 3832 2034  0.00 0.00 1/82 4
00000800: 3931 0a1b 5b37 6d20 2050 4944 2020 5050  91..[7m  PID  PP
00000810: 4944 2055 5345 5220 2020 2020 5354 4154  ID USER     STAT
00000820: 2020 2056 535a 2025 5653 5a20 4350 5520     VSZ %VSZ CPU
00000830: 2543 5055 2043 4f4d 4d41 4e44 1b5b 6d0a  %CPU COMMAND.[m.
00000840: 2020 3437 3320 2020 2020 3020 3939 3920    473     0 999
00000850: 2020 2020 2053 2020 2020 3535 3538 3020       S    55580
00000860: 2020 3125 2020 2032 2020 2030 2520 7265    1%   2   0% re
00000870: 6469 732d 7365 7276 6572 202a 3a36 3337  dis-server *:637
00000880: 390a 2020 3438 3920 2020 2020 3020 726f  9.  489     0 ro
00000890: 6f74 2020 2020 2053 2020 2020 2034 3339  ot     S     439
000008a0: 3620 2020 3025 2020 2032 2020 2030 2520  6   0%   2   0%
000008b0: 746f 700a 2020 2033 3020 2020 2020 3020  top.   30     0
000008c0: 726f 6f74 2020 2020 2053 2020 2020 2031  root     S     1
000008d0: 3732 3020 2020 3025 2020 2030 2020 2030  720   0%   0   0
000008e0: 2520 2f62 696e 2f73 680a 2020 2032 3920  % /bin/sh.   29
000008f0: 2020 2020 3020 726f 6f74 2020 2020 2052      0 root     R
00000900: 2020 2020 2031 3632 3020 2020 3025 2020       1620   0%
00000910: 2032 2020 2030 2520 746f 700a 2020 2020   2   0% top.
00000920: 3120 2020 2020 3020 3635 3533 3520 2020  1     0 65535
00000930: 2053 2020 2020 2020 3936 3820 2020 3025   S       968   0%
00000940: 2020 2033 2020 2030 2520 2f70 6175 7365     3   0% /pause
00000950: 0d2f 7061 7573 650d 2020 3020 726f 6f74  ./pause.  0 root
00000960: 2020 2020 2053 2020 2020 3131 3338 3820       S    11388
00000970: 2020 3025 2020 2032 2020 2030 2520 6e67    0%   2   0% ng
00000980: 696e 783a 206d 6173 7465 7220 7072 6f63  inx: master proc
00000990: 6573 7320 6e67 696e 7820 2d67 2064 6165  ess nginx -g dae
000009a0: 0a20 2020 3330 2020 2020 2030 2072 6f6f  .   30     0 roo
000009b0: 7420 2020 2020 5320 2020 2020 3137 3230  t     S     1720
000009c0: 2020 2030 2520 2020 3020 2020 3025 202f     0%   0   0% /
000009d0: 6269 6e2f 7368 0a20 2020 3239 2020 2020  bin/sh.   29
000009e0: 2030 2072 6f6f 7420 2020 2020 5220 2020   0 root     R
000009f0: 2020 3136 3230 2020 2030 2520 2020 3220    1620   0%   2
00000a00: 2020 3025 2074 6f70 0a20 2020 2031 2020    0% top.    1
00000a10: 2020 2030 2036 3535 3335 2020 2020 5320     0 65535    S
00000a20: 2020 2020 2039 3638 2020 2030 2520 2020       968   0%
00000a30: 3320 2020 3025 202f 7061 7573 650d 6f70  3   0% /pause.op
00000a40: 0a20 2020 2031 2020 2020 2030 2036 3535  .    1     0 655
00000a50: 3335 2020 2020 5320 2020 2020 2039 3638  35    S      968
00000a60: 2020 2030 2520 2020 3320 2020 3025 202f     0%   3   0% /
00000a70: 7061 7573 650d 650d 2030 2520 2f70 6175  pause.e. 0% /pau
00000a80: 7365 0d20 2030 2520 6177 6b20 7b70 7269  se.  0% awk {pri
00000a90: 6e74 2024 284e 4620 2d20 3129 202a 2031  nt $(NF - 1) * 1
00000aa0: 3030 307d 0a20 2034 3232 2020 2020 3330  000}.  422    30
00000ab0: 2072 6f6f 7420 2020 2020 5320 2020 2020   root     S
00000ac0: 3136 3034 2020 2030 2520 2020 3220 2020  1604   0%   2
00000ad0: 3025 2067 7265 7020 5472 616e 7366 6572  0% grep Transfer
00000ae0: 7265 640a 2020 3432 3120 2020 2033 3020  red.  421    30
00000af0: 726f 6f74 2020 2020 2053 2020 2020 2031  root     S     1
00000b00: 3030 3020 2020 3025 2020 2030 2020 2030  000   0%   0   0
00000b10: 2520 7363 7020 2d76 2032 2d35 3536 3164  % scp -v 2-5561d
00000b20: 6666 6539 3030 302d 3535 3631 6530 3030  ffe9000-5561e000
00000b30: 3730 3030 0000 0000 0000 0000 0000 0000  7000............
```

```
Mem: 902720K used, 6806400K free, 734120K shrd, 980K buff, 782384K cached
CPU:    0% usr    0% sys    0% nic 100% idle    0% io    0% irq    0% sirq
Load average: 0.00 0.00 0.00 1/82 491
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
  473     0 999      S    55580   1%   2   0% redis-server *:6379
  489     0 root     S     4396   0%   2   0% top
   30     0 root     S     1720   0%   0   0% /bin/sh
   29     0 root     R     1620   0%   2   0% top
    1     0 65535    S      968   0%   3   0% /pause

   ..     0 root     S    11388   0%   2   0% nginx: master process nginx -g dae
   30     0 root     S     1720   0%   0   0% /bin/sh
   29     0 root     R     1620   0%   2   0% top
    1     0 65535    S      968   0%   3   0% /pause

  ...    .. ....     .     ....   ..   .   0% awk {print $(NF - 1) * 1000}
  422    30 root     S     1604   0%   2   0% grep Transferred
  421    30 root     S     1000   0%   0   0% scp -v 2-5561dffe9000-5561e0007000
```

**Figure 2: Extraction of Confidential Information from Memory Dump of Top**

unauthorized access and memory leakage from another container. Revealing the content of confidential memory, whether it contains secrets or sensitive information, represents a breach of the memory confidentiality guarantee provided by confidential computing. Such a breach carries significant security risks, including data breaches, identity theft, privacy violations, intellectual property theft, and a variety of possible cyber-security incidents.

## 4.2 Obtaining Runtime Metrics Information

This attack allows an adversary on the host to acquire runtime metrics information from within a confidential VM. Such statistics should ideally only be accessible from within the VM [20]. With memory encryption protection, the host should not have the ability to retrieve such data from outside, even when employing virtual machine introspection (VMI) [18].

To obtain runtime metrics from a confidential VM, we conducted an attack by sending the `GetMetrics` API command to the `kata-agent` operating within a confidential VM. Like the previous attack scenario, we first obtained the `sandbox` ID from the host file system and then invoked the `GetMetrics` API command. As a result of the API command, we retrieved a substantial amount of runtime metrics information. This information includes process status, CPU time, disk statistics, memory information, network device information, virtual memory statistics, and other information.

The accessibility of such information by an adversary can lead to various security implications, including the ability to conduct reconnaissance for vulnerabilities, engage in potential side-channel attacks, establish covert communication channels, identify potential targets for DoS attacks, etc.

## 4.3 Pausing/Resuming/Removing Containers

This attack is in the category of execution tampering, illustrating that adversaries can exploit specific `kata-agent` APIs to manipulate confidential containers, including actions like pausing, resuming, or even completely removing them. The goal is to cause service disruption, data inconsistencies, data loss, and service unavailability. It is different from the traditional availability attacks, which are more coarse-grained and potentially lead to the termination of the entire VM. In contrast, this attack operates on a finer-grained level, exerting control over the execution of individual containers within a VM.

Similar to the attack scenarios described before, we assume the presence of an operational container within a confidential VM, referred to as a *victim* container. We initiated the attack by first extracting the `sandbox` and `container` IDs from the host file system. Subsequently, we performed `pause`, `resume`, and `removal` operations on the *victim* container using the `PauseContainer`, `ResumeContainer`, and `RemoveContainer` APIs. We issued these API commands and got return values indicating the success or failure of each operation. We also successfully paused and resumed the execution of the *victim* container and subsequently removed it, as confirmed by the status message.

It is important to note that these container manipulations occur within the confidential VM. They remain hidden from the host control plane, which includes components like `containerd` or any higher-level orchestration management systems. As a result, the host control plane may still display outdated container status, making it challenging for the workload owners to detect any tampering attempts or send commands to the incorrect or non-existent target containers, particularly when all control has to pass through the host control plane. This lack of visibility can further complicate the tasks of container management and monitoring.

This form of execution tampering attack carries significant security implications. Illegitimate pause and resume actions can disrupt services, lead to data loss, cause inefficient resource allocation, introduce security vulnerabilities, and increase operational overhead. Unauthorized removal of container workloads can compromise the availability and integrity of services.

## 4.4 Setting Bad Date and Time

Many applications rely on accurate time sources to function properly. However, malicious modifications to date and time settings can disrupt the execution of processes and lead to significant security vulnerabilities. For instance, security mechanisms, such as public key certificates and digital signatures, depend on precise time information. Similarly, time-based authentication tokens, event logging, scheduled tasks, backup services, and software licensing mechanisms all rely on accurate timestamps for their operations. Even monitoring and auditing tasks can be compromised if the system time is tampered with.

CoCo exposes the `SetGuestDateTime` API to the host. Adversaries can exploit this API to maliciously alter the date and time settings within a VM. To illustrate the impact of this attack, we established a Transport Layer Security (TLS) channel using TLS certificates to communicate with a container from a client program. Subsequently, we exploited the `SetGuestDateTime` API to manipulate the VM's system time, adjusting it to an old timestamp (*e.g.*, January 1, 1970). Due to this alteration, any subsequent attempts to establish a TLS channel between the TLS server and the client program failed, primarily because the client certificate's validity period was out of range from the current time. This serves as a clear example of the disruptive consequences that can arise from tampering with the date and time settings.

## 4.5 Writing Sensitive Files

This attack allows an adversary to conduct privilege escalation attacks by adding new or overwriting existing files in the VM's `/run/kata-containers` directory. Since this directory stores all the runtime data associated with managing and running containers in the VM, writing into this directory has severe security ramifications such as tampering with container execution, escalating privileges, and adding or dropping capabilities.

To demonstrate this attack, we first created a confidential VM and obtained its `sandbox` ID. After that, we utilized the `kata-agent-ctl` tool to invoke the `CopyFile` API from the host to overwrite the `readonly` flag of the VM's `rootfs` from `true` to `false`. We made this change by overwriting the `config.json` configuration file located in the `/run/kata-containers/<sandbox_ID>/` directory within the VM. After that, we manually inspected the configuration file and verified the reflected change. In addition to changing the configuration files, an adversary can target a specific file or replace any executables or libraries under the same directory.

## 5 DEFENSE APPROACH

To address the security gap discussed in Section 3, we propose a redesign of CoCo's control interface with a focus on minimizing the attack surface. This redesign is guided by three security principles:

**Realignment of Threat Model.** Our defense approach must align with the same threat model as confidential computing, which assumes that the host software stack is untrusted. Consequently, the host should not have the ability to access private data or exert control over operations within protected domains. It is important to note that, similar to confidential computing, our approach does not address availability threats like DoS attacks.

**Separation of Responsibilities.** We advocate for a clear separation of responsibilities between the host-side and the owner-side controllers. The host-side controller should focus solely on the allocation and recycling of compute resources for confidential VMs and be compatible with the orchestration layer. It should be restricted from exercising other capabilities that would violate the threat model of confidential computing. In contrast, we should empower the workload owners to directly manage the deployed pod and containers. This requires a thorough examination of the security properties of each API endpoint, categorizing them for different controllers.

**Protection of Remote Control.** We emphasize the importance of ensuring that the owners of the container workloads are authorized to manage containers. Additionally, the communication tunnel used for sending commands from authorized workload owners should have end-to-end protection to protect against unauthorized access or tampering.

## 5.1 Design Principles

Our design involves the partitioning of CoCo's API endpoints to cater to two distinct controllers: the *host-side* and the *owner-side*. Here, we describe the responsibilities of these controllers across different phases in the life cycle of containers:

- *Initialization Phase*: during this phase, the host-side controller takes charge of provisioning resources required to launch confidential VMs. The `kata-agent` in the VM then performs attestation to the KBS to obtain keys for unpacking container images and deploying pods and containers.
- *Runtime Phase*: in this phase, the workload owner steps in to take control. The owner authenticates and establishes an end-to-end secure tunnel for managing container deployments. The owner possesses a local control plane that enables direct communication with the `kata-agent`, giving them control over their containers.
- *Termination Phase*: once containers have completed their tasks, the host-side controller is responsible for the orderly destruction of confidential VMs and the recycling of all associated resources.

We aim to avoid expanding the TCB when deploying workloads on host systems. The design does not introduce any additional components to host systems. Furthermore, it also maintains a flexible deployment model that is compatible with the existing cloud-native orchestration layer. Now let us delve into the details.

**Partitioning of CoCo's Control Interface.** We have categorized the `kata-agent` API endpoints based on their intrinsic security properties concerning *confidentiality*, *integrity*, and *availability*. In Table 1, we provide a list of the `kata-agent`'s APIs with *partitioning labels*. Note that some APIs in the table are marked as `Not Supported`, such as `AddSwap` and `MemHotplugByProbe`. It is because the corresponding features are not yet supported for confidential VMs, thus these APIs are disabled by default in CoCo.

We have identified a group of APIs that have distinct functional boundaries, allowing us to allocate them exclusively to either the host-side or the owner-side:

`Host-Exclusive`: APIs related to the availability of host resources are designated for the host-side. These APIs provide access to and

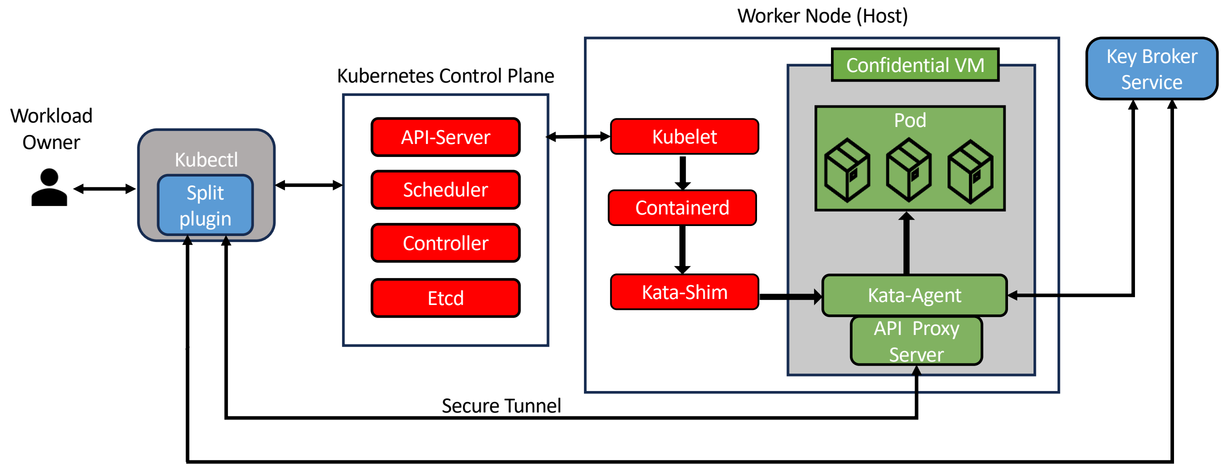


**Figure 3: Partitioned Architecture of Confidential Containers Control Plane**

usage of host resources, such as the confidential VM and host networking. An example is the `CreateSandbox`, which initializes the `sandbox` environment inside a VM after the VM boots. Another example is the `DestroySandbox`, which destroys the `sandbox` environment during the termination phase, terminating all containers in the `sandbox` and notifying the `kata-agent` to exit.

`Owner-Exclusive` : APIs affecting the confidentiality or integrity of the owner's workloads are reserved for the owner-side. The idea is that security-sensitive APIs are restricted to the owner-side only. For example, workload owners can invoke the `ExecProcess` to create and run a process inside of a container. They can also invoke `RemoveContainer` to remove a container within a VM.

We also recognize that certain APIs, whose scopes overlap with both the host's operational and availability objectives and the owner's confidentiality and integrity requirements, must be separated. Separating an API entails providing a version for the host-side and another for the owner-side. For these separated APIs, we introduce three types of labels:

`Shared` : for shared APIs, the same functionality is available to both the host-side and the owner-side. These APIs are designed to allow visibility of host-allocated resources or notify the owner-side of events. Examples include `ListInterface` and `ListRoute`, which provide both the host-side and the owner-side with network configuration information of the confidential VM.

`Sanitized` : in the case of sanitized APIs, we ensure that the host-side variant does not violate the owner's security requirements. The input or output of an API invocation is sanitized to prevent the host-side from getting sensitive information from or tampering with owner workload execution. Examples include the `GetGuestDetails` and `SignalProcess`. When `GetGuestDetails` is invoked from the host-side, its output is filtered to provide only host-relevant metrics information. Similarly, the input of the `SignalProcess` is restricted to signaling the initial container associated with the `sandbox`, preventing misuse by the host-side to terminate any container process running in the confidential VM.

`Switch` : Switch APIs dynamically transition their roles from `Host-Exclusive` to `Owner-Exclusive` . These APIs are accessible only by the host during the initialization phase, after which the control shifts to the owner. This labeling is designed to align with CoCo's execution model. `CreateContainer` and `StartContainer` are two examples of Switch APIs. In CoCo, the host controller utilizes these APIs to establish an initial (immutable) container within the sandbox VM during initialization. This initial container is integrated into the guest file system and undergoes verification during startup. Subsequently, after attestation, we restrict access to these APIs from the host-side controller, permitting invocation exclusively by the owner-side controller.

**Bifurcation of Control Plane.** Based on the partitioned control interface, we introduce two separate controllers: the *host-side controller* and the *owner-side controller*, which interact with the `kata-agent` using distinct channels.

- *Host-Side Controller.* The host-side controller continues to utilize the existing gRPC over `vsock` channel, but its capabilities are substantially limited. It is now responsible only for allocating and recycling compute resources.
- *Owner-Side Controller.* The owner-side controller establishes a secure tunnel to the `kata-agent` within the VM, allowing the workload owner to securely transmit commands to the `kata-agent`. This setup ensures that requests and responses are protected during their transmission between the owner-side and the `kata-agent`. They are only in plaintext at the secure tunnel's endpoints: the owner-side and the confidential VM where the `kata-agent` operates. As a result, the commands issued by the owner no longer traverse the unprotected gRPC over `vsock` channel on the host system. This configuration has the added benefit of bypassing not only the untrusted host container runtime components (*e.g.,* `kubelet`, `containerd`, and `kata-shim`), but also the orchestration layer (*e.g.,* `Kubernetes` control plane).

**Secure Tunnel Establishment.** The owner secret delivery process for establishing the secure tunnel between the owner-side controller and the `kata-agent` is integrated into the remote attestation process with the KBS, a trusted entity responsible for authenticating the owner-side controller and releasing owner secrets to CoCo. A TLS client certificate of the owner-side controller is pre-registered with the KBS before an owner initiates any workloads. During sandbox initialization, the `kata-agent` requests the owner secret from the KBS, providing a signed attestation report containing the guest measurements. The KBS address is specified as a guest kernel option and is included in the measurements. Upon successful verification, the KBS generates ephemeral public and private key pairs for the `kata-agent`'s proxy server, returning them as the owner secret, including the TLS client certificate for establishing the gRPC TLS connection.

### 5.2 System Architecture

We describe the components of the partitioned control plane and their interaction using a `Kubernetes` cluster deploying confidential containers as shown in Figure 3.

**Resource Provisioning.** A workload owner uses the `kubectl` cluster management tool [34] as the host-side controller to provision resources. To do so, `kubectl` interacts with the `Kubernetes` API-Server to request a confidential VM on a host system. The `Kubernetes` API-Server forwards the request to the `kubelet` process running on a host system. The `kubelet` process [30] interacts with the `containerd` and `kata-shim` to launch a confidential VM, which then starts the `kata-agent`. As part of its startup process, the `kata-agent` retrieves the owner secret from the KBS and starts its *API proxy server* to service API requests that come from the workload owner through a secure tunnel.

**Workload Management.** We have developed a `kubectl` plugin [29], called `split`, for workload owners to manage their containers inside confidential VMs. Acting as an owner-side controller, the `split` plugin sends requests and receives responses to/from `kata-agent`'s *API proxy server* by establishing a secure tunnel. Note that the workload owner pre-registers the `split` plugin's TLS client public key certificate with the KBS before starting the VM. This certificate is provided as part of the owner secret to the `kata-agent` and allows the *API proxy server* to authenticate connection requests from the `split` plugin. After the initialization phase, the `split` plugin retrieves communication parameters from the KBS using the `sandbox` ID. These communication parameters consist of the public key certificate of the generated key pair for *API proxy server* and the IP address of the VM running the `kata-agent`'s *API proxy server*. The `split` plugin uses these parameters to establish the secure tunnel with the `kata-agent`. During the runtime phase, the workload owner executes `split` plugin commands to deploy containers in the VM. When creating and starting a container, the `split` plugin generates a sequence of API requests consisting of `PullImage`, `CreateContainer`, and `StartContainer` to the `kata-agent`.

### 5.3 Implementation

For our implementation, we extended the CoCo branch (`CCv0`) [7], embedding a gRPC TLS server into the `kata-agent` to handle API requests from the owner side. Since the `kata-agent` is coded in Rust [36], we leveraged `tonic` [46], a `Rust` package providing a high-performing gRPC over `HTTP/2` framework with TLS support. The extension to the agent's codebase has 900 lines of `Rust`, with an additional 2394 lines of code automatically generated for client and service stubs from the `tonic` build. Additionally, we developed the `kubectl split` plugin [29] based on the `kata-agent-ctl` [47], a low-level utility tool interacting with the `kata-agent`. Our version of the tool establishes a TLS connection with the `kata-agent` for sending API requests. The plugin code has 2300 lines of `Rust`.

## 6 DEFENSE EVALUATION

We conducted a comprehensive evaluation of our proposed defense approach, examining both its security and performance implications. In the security assessment, we provide a detailed overview of the steps taken to mitigate the attacks, highlighting the effectiveness of our defense mechanisms. Additionally, in the performance measurement, we present a comparative analysis between our implementation and the original design of CoCo, showcasing any performance trade-offs resulting from our approach.

### 6.1 Security Assessment

In Section 4, we discuss five attacks targeting the `kata-agent` API endpoints exposed to the host-side controller for managing container workloads. Our defense addresses these vulnerabilities by removing or sanitizing these APIs from the host control plane, allowing full control through an owner-side controller. Below, we detail the mitigation strategies for each attack respectively.

**Attack §4.1:** This attack leverages the `PullImage`, `CreateContainer`, and `StartContainer` APIs to deploy an *attacker* container within the same sandbox as the *victim* container, facilitating memory dumping. To counter this attack, we designate `PullImage` as `Owner-Exclusive`, only allowing the workload owner to request the image pull. `CreateContainer` and `StartContainer` are classified as `Switch`, limiting host-side controller access during initialization phase. In the following phases, only the owner-side controller can invoke these APIs to launch new containers within a sandbox, preventing attackers from injecting malicious containers.

**Attack §4.2:** This attack exploits the `GetMetrics` API to obtain runtime metrics information of the sandbox. This API has been classified as `Sanitized`. We still allow the host-side controller to invoke this API while sanitizing returned results to remove sensitive information. The owner-side controller can obtain original results. This approach thwarts attackers from obtaining runtime metrics from the host.

**Attack §4.3:** Attackers exploit `PauseContainer`, `ResumeContainer`, and `RemoveContainer` APIs to manipulate containers within the sandbox. By designating these APIs as `Owner-Exclusive`, we prevent host-side controller invocation, thereby thwarting unauthorized container manipulation by adversaries.

**Attack §4.4:** This attack utilizes the `SetGuestDateTime` API to modify the system time, thwarting attempts to establish a TLS channel. We classify the `SetGuestDateTime` as `Owner-Exclusive`, thereby preventing unauthorized modification of date and time within the sandbox by attackers on the host.

**Attack §4.5:** This attack exploits the `CopyFile` API to write files into a confidential VM's `/run` directory. By designating this API as

**Table 3: The pod-level performance was evaluated using the `crictl` tool, while the container-level performance was assessed using the `kata-agent-ctl` tool and the `kubectl` `Split` plugin separately. The average execution times for container-level API invocations are presented in two columns: `Total` and `API`. A `Total` column includes both communication time and API processing time, while an `API` column exclusively denotes API processing time. All time values are expressed in milliseconds.**

| **Image name** | **alpine** | | | | **nginx** | | | | **redis** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Image ID** | b2aa39c304c2 | | | | a8758716bb6a | | | | bdff4838c172 | | | |
| **Image size** | 7.05 MB | | | | 187 MB | | | | 138 MB | | | |
| **Pod-Level** | CoCo | | Split | | CoCo | | Split | | CoCo | | Split | |
| `runp` | 7825.2 | | 7939.2 | | 7978.3 | | 7891.6 | | 7986.5 | | 7986.9 | |
| `stopp` | 469.1 | | 501.2 | | 533.7 | | 522.7 | | 529.7 | | 597.4 | |
| `rmp` | 116.6 | | 111.5 | | 114.4 | | 117.3 | | 112.1 | | 113.6 | |
| **Container-Level** | Total | API | Total | API | Total | API | Total | API | Total | API | Total | API |
| `Check` | 2.9 | 5.5E-05 | 19.2 | 6.1E-05 | 3 | 5.5E-05 | 18.9 | 5.7E-05 | 2.9 | 5.4E-05 | 19.3 | 5.4E-05 |
| `CopyFile` | 7.6 | 0.038 | 19.5 | 0.036 | 7.5 | 0.032 | 19.8 | 0.039 | 7.5 | 0.034 | 19.5 | 0.036 |
| `CreateContainer` | 20.9 | 15.107 | 40.7 | 16.54 | 20.6 | 14.71 | 41.1 | 16.4 | 19.9 | 14.33 | 38.3 | 13.81 |
| `GetGuestDetails` | 3 | 0.032 | 19 | 0.02 | 3 | 0.026 | 19.1 | 0.019 | 2.8 | 0.032 | 19.2 | 0.018 |
| `GetMetrics` | 8.4 | 4.143 | 24.6 | 4.626 | 8.2 | 4.32 | 23.9 | 3.87 | 8.6 | 4.29 | 25.1 | 5.41 |
| `ListInterfaces` | 3 | 0.117 | 19.5 | 0.154 | 3 | 0.108 | 19.2 | 0.153 | 3 | 0.113 | 19.6 | 0.149 |
| `ListRoutes` | 3 | 0.171 | 19.1 | 0.194 | 3 | 0.164 | 19.1 | 0.204 | 3 | 0.15 | 19.2 | 0.206 |
| `PauseContainer` | 3 | 0.028 | 19.3 | 0.037 | 3 | 0.031 | 19.4 | 0.031 | 3 | 0.021 | 19.6 | 0.034 |
| `PullImage` | 719.5 | 714.02 | 679.70 | 650.8 | 2714.5 | 2708.4 | 2851 | 2833.37 | 2189.2 | 2183.75 | 2252.1 | 2222.58 |
| `RemoveContainer` | 3.1 | 0.553 | 19.8 | 0.727 | 3.8 | 0.584 | 19.9 | 0.785 | 3.3 | 0.585 | 20.1 | 0.695 |
| `ReseedRandomDev` | 3 | 0.024 | 19.1 | 0.032 | 3 | 0.029 | 19.1 | 0.029 | 3 | 0.028 | 19.3 | 0.03 |
| `ResumeContainer` | 3 | 0.015 | 19.1 | 0.065 | 3.1 | 0.017 | 19.1 | 0.029 | 3 | 0.017 | 19.1 | 0.025 |
| `SignalProcess` | 3 | 0.067 | 19.3 | 0.094 | 3 | 0.04 | 18.8 | 0.045 | 3 | 0.068 | 19.2 | 0.051 |
| `StartContainer` | 3.1 | 0.111 | 20.9 | 0.864 | 3 | 0.11 | 20.8 | 0.235 | 3.1 | 0.108 | 20.4 | 0.136 |
| `StatsContainer` | 5.4 | 1.95 | 20 | 0.55 | 3.7 | 0.49 | 20.6 | 0.57 | 4 | 1.228 | 20.3 | 0.545 |
| `Version` | 3 | 3.7E-04 | 18.9 | 6.0E-05 | 3 | 3.4E-04 | 19.1 | 5.9E-05 | 3 | 2.3E-04 | 18.9 | 6.5E-05 |
| `WaitProcess` | 3 | 0.047 | 19.2 | 0.048 | 3 | 0.045 | 18.8 | 0.087 | 3 | 0.062 | 19 | 0.067 |

`Owner-Exclusive`, host write access to the directory is restricted, mitigating the attack.

Please refer to Table 1 for the complete list of API endpoints classification. By labeling certain APIs as `Owner-Exclusive`, `Sanitized`, and `Switch`, we implement different API invocation policies, thereby effectively minimizing host-side attack surface and mitigating potential attacks.

## 6.2 Performance Measurement

For our performance evaluation, we used the same hardware setup as the security experiments. The detailed configuration specifications are available in Section 4. We deployed three representative container workloads — `alpine`, `nginx`, and `redis` — and measured execution times for pod-level and container-level operations in two distinct modes: `CoCo` and `Split`. In the `CoCo` mode, serving as our baseline for comparison, we maintain the standard configurations of CoCo. Conversely, in the `Split` mode, we enable all security features described in Section 5 to evaluate their impact on performance.

Pod-level operations include the initiation, termination, and removal of pods. To execute these operations, we utilized the Container Runtime Interface (CRI) command-line tool, `crictl` [8], specifying `Kata` as the runtime. This tool facilitated the creation of a new pod, initiating a confidential VM and establishing its sandbox environment within the running confidential VM. We ran a `crictl`, a KBS, and an image registry on the same host where the workloads were deployed to ensure consistency and eliminate network variability. Pod-level operations were measured due to the additional initialization steps required in the `Split` mode, wherein the `kata-agent` retrieves the owner secret from the KBS and initiates its API proxy server — an overhead not present in the `CoCo` mode.

Container-level operations include the invocations of `kata-agent` APIs. In the `CoCo` mode, we used the `kata-agent-ctl` [47] tool to send requests over `vsock`, the same way as in the original CoCo's design. In the `Split` mode, we used the `split` plugin to send API requests to the `kata-agent` via a secure channel.

We executed each workload ten times and computed the averages of `crictl` commands (namely, `runp`, `stopp`, and `rmp`) and `kata-agent` API invocations.

In Table 3, we first display the average execution times for the pod-level operations in `CoCo` and `Split` modes. We observed that the times for staring, stopping, and removing a pod in both modes are relatively similar. The ratio of `Split` over `CoCo` execution time is close to one for most operations, suggesting that the overhead introduced by the `kata-agent`'s API Proxy has a negligible performance impact on managing pods.

Subsequently, we present the execution times of container-level operations by invoking the `kata-agent` APIs, which have been modified to enable owner control. We break down the average execution times for communication and API processing. They are presented in two columns: `Total` and `API`. A `Total` column includes both communication time and API processing time, while an `API` column exclusively denotes API processing time. Computing the geometric means of the `Total` columns using `CoCo` as the baseline yields values of 4.69, 4.76, and 4.8 when deploying the respective images `alpine`, `nginx`, and `redis`. These results suggest that `CoCo` mode is 4.69 to 4.8 times faster than `Split` mode. Conversely, our analysis of the API execution times and their corresponding proxy APIs shows minimal performance overhead when executing the proxy APIs. This finding is supported by the computation of the geometric means for the API columns, using `CoCo` as the baseline, yielding values of 1.10, 1.07, and 0.96 for the respective images `alpine`, `nginx`, and `redis`.

It is important to note that these performance overheads are linked solely to each container management command. These overheads are only incurred when specific commands are invoked and do not impact the runtime performance of the container workloads.

## 7 DISCUSSION

Here, we delve into the implications of our proposed design on the existing and upcoming features of CoCo, while also outlining potential avenues for future research.

### 7.1 Debugging

Given that confidential computing is specifically engineered to protect data-in-use, it inherently restricts mechanisms that expose data to external processes on the host. This limitation applies to various debugging tools, ranging from simple logging or tracing mechanisms to more complex ones like VMI. Allowing protected data to be exposed through such mechanisms would directly undermine the core purpose of confidential computing and, therefore, cannot be permitted.

Our proposed design effectively disables the interface exposed to the untrusted host for debugging purposes. Conversely, it offers a secure pathway for workload owners to remotely retrieve container logs and execute commands for troubleshooting. If without our design, the only alternative would be to block the corresponding features entirely, rendering it nearly impossible to debug malfunctioning containers in confidential VMs.

### 7.2 Data Sharing

Similarly, sharing data between containers using mechanisms like shared memory or inter-process communication (IPC) contradicts the principles of confidentiality, which prioritize keeping data private to its owner. This does not introduce a new restriction about CoCo or Kata, as containers are already isolated in different VMs. Only containers within the same pod (and thus, within the same VM) can effectively share memory via the kernel.

In theory, it is also possible to share memory pages with the host via hypervisor-level mechanisms, as seen, for instance, when using `virtiofs`. However, such mechanisms should be avoided whenever possible, as they may expose owner data to the host, similar to unencrypted serial consoles.

Our proposed design maintains the existing mechanisms concerning shared memory. However, it securely enables owner-oriented data sharing features such as `CopyFile`. This only allows workload owners, rather than the host, to securely copy data to and from containers.

### 7.3 I/O Operations

When a container wants to perform I/O operations, it has to rely on hardware provided by the host, such as a network interface card or a disk controller. This entails a level of data sharing between the guest and host. The straightforward way to achieve that objective is to use non-protected data pages in the guest memory. Existing TEE technologies that rely on memory encryption all have mechanisms to designate specific pages as *shared*. For instance, both AMD SEV and Intel TDX allow guest VM owners to convert memory pages from encrypted to shared by changing a specific bit in the guest physical address (GPA). To preserve confidentiality, data on such pages should still be encrypted by the users, typically using TLS for networking or Linux Unified Key Setup (LUKS) for storage.

This aspect of I/O operations is already taken into account by the existing CoCo implementation, and our proposal does not affect its security. However, as discussed above, we do markedly improve usability by facilitating owner access to encrypted I/O facilities, exposing through existing channels such as `ExecProcess` or `CopyFile`. The alternative in the current CoCo implementation is to set up an `ssh` access to the container, which adds complexity to the setup and must be done ahead of time even if such services are only a contingency plan and not part of the core services provided by the container.

Starting from the Hopper architecture, Nvidia has initiated support for confidential computing [13] on GPUs. The process involves a GPU being integrated into the trust boundary following authentication by a confidential VM. The current approach relies on a software-based bounce-buffer technique, transferring data through encrypted staging buffers in shared memory. As a result, this method incurs limitations on CPU-GPU bandwidth and introduces additional latency overhead during data transfer. To further enhance I/O performance, the development of trusted I/O [4, 25] is underway, built upon multiple industry standards, including the TEE Device Interface Security Protocol (TDISP), Integrity and Data Encryption (IDE), and Secure Protocol and Data Model (SPDM). These efforts aim to support fast and secure I/O between confidential VMs and TEE-aware physical devices.

Such mechanisms will enable guests to verify the attestation of devices and facilitate Direct Memory Access (DMA) to encrypted memory. Consequently, this approach saves on the costly conversion of memory pages and eliminates the need for additional data copies. Our design will seamlessly leverage such trusted I/O acceleration once it becomes commonly available, by using a common software stack that will be modified as needed to take advantage of such devices.

### 7.4 Orchestration Interaction

A straightforward method to implement containers with confidential computing is to construct the entire cluster using confidential VMs. Instead of initiating a confidential VM per pod (as seen in CoCo), one could also use multiple confidential VMs to serve as control and worker nodes within a traditional cluster setup [9]. This approach offers protection for both container workloads and orchestration operations.

However, a drawback of this approach is that it shifts the entire responsibility for operating and managing the clusters onto end users. This is incompatible with the cloud-managed `Kubernetes-as-a-service` model, where users rely on cloud providers to manage the cluster resources. Many end users may also lack the expertise required for managing clusters, making this approach potentially more expensive and less flexible.

The primary advantage of the CoCo approach lies in the ability for the cloud to still manage the control nodes and control the compute resources for a cluster. This allows end users to ensure that their workloads run within protected domains without having to worry about orchestration. Splitting resources between confidential and non-confidential sections is much more cost-effective. The orchestration components in the control plane do not consume confidential resources and can thus run on non-confidential nodes. However, it is crucial to note that the control plane is no longer considered trusted, and the data it contains is assumed to be accessible by a rogue system administrator. This necessitates additional design changes proposed in this paper.

Splitting the API responsibilities between host-side and owner-side controllers, as we propose, makes this possible. However, this implies that the entire orchestration layer is outside of the owner's control. The value added by this orchestration layer includes automatic provisioning of resources, auto-scaling, and transparent fail-over, all of which should work with our design, but are not controlled by it.

## 8 RELATED WORK

In this section, we begin by reflecting on recent developments in systems that utilize confidential computing to protect containers. Following this, we delve into various research efforts concentrating on attacks and defenses on general container security.

### 8.1 Protecting Containers with Confidential Computing

Confidential computing aims to create a secure execution domain that is isolated from other host components. One of the ongoing debates in research revolves around the granularity of this protected domain, whether it should encompass individual functions, libraries, applications, or even entire virtual machines. Striking a balance between ensuring trustworthiness and maintaining usability is the key challenge in this regard.

Intel SGX [37] secures a specific memory region within an application's address space, requiring developers to decide which part of their application should be enclosed for protection. However, SGX's protection granularity is not conducive to running unmodified applications or containers, which typically rely on dynamic linking with multiple libraries and require operating system call services. To address this limitation, researchers have explored alternative approaches. These include the use of a shield layer, as seen in `Scone` [5] and `Panoply` [44], or the incorporation of a library operating system (LibOS), exemplified by `Graphene` [49] and `Occlum` [43], within the enclave.

Recent advancements in confidential computing technologies, such as AMD SEV [27, 28, 42], Intel TDX [26], IBM SE [23], PEF [22], and ARM CCA [32], tend to embrace a protection granularity at the virtual machine level. This approach significantly eases the challenge of running unmodified applications or containers, as all the necessary libraries and system services are enclosed within the same secure domain. Nevertheless, this shift can raise discussions about whether the TCB has become excessively large.

The CoCo project begins by harnessing VM-based confidential computing to protect container workloads within confidential VMs. Recently, CoCo has extended its support to include process-based isolation [11] using SGX and LibOS [43, 49]. CoCo's objective is to integrate with the `Kubernetes-as-a-Service` (or `managed Kubernetes`), where cloud service providers (CSPs) host the `Kubernetes` clusters. However, as discussed in this paper, the introduction of confidential computing necessitates a restriction on the capabilities of the CSPs. They should only be able to allocate and recycle compute resources, following the threat model of confidential computing. They should not have the privilege to access private data or exert control over the execution within the protected domains.

In a related development, Constellation [9] is another initiative that leverages confidential computing for container deployment. This approach seeks to secure both the worker nodes and the `Kubernetes` control plane nodes within confidential VMs. Consequently, it is suitable for scenarios involving the deployment of self-managed `Kubernetes` clusters in the cloud. This deployment style offers enhanced security benefits since all nodes operate within protected domains. However, it also requires users to possess the expertise to manage the cluster independently.

Our work focuses on the security implications of CoCo's approach of integrating confidential computing and Kata Containers. Our proposed design for splitting CoCo's control interface can both align with the threat model of confidential computing and support cloud-managed `Kubernetes` clusters.

### 8.2 Attacks/Defenses in Container Security

In addition to industry solutions for securing containers with confidential computing, there is a significant body of research focused on identifying security vulnerabilities and proposing enhanced security features for containers. Here is a brief overview of some notable research contributions in this domain.

Gupta [21] conducted a comparative analysis of the security of containers and virtual machines. Bui [6] analyzed the security aspects of Docker containers, with a specific focus on their isolation mechanisms and interactions with Linux kernel security features. Grattafiori *et al.* [19] summarized various potential vulnerabilities associated with containers. Luo *et al.* [35] identified potential covert channels in Docker that could lead to cross-container information leaks. Gao *et al.* [15, 17] carried out security studies on Linux `namespaces` and explored how information could leak through memory-based pseudo file systems. Lei *et al.* [31] introduced a system named `SPEAKER`, designed to reduce the number of system calls available to containerized applications. Lin *et al.* [33] conducted a systematic evaluation of container security using real-world exploits and found that many of these exploits could succeed within containers configured with default settings. Sun *et al.* [45] developed two security `namespaces` that enable autonomous security control for containers. Gao *et al.* [16] explored methods to break the performance confinement of `cgroups`, potentially impacting the computing and I/O performance of co-resident containers. Xiao *et* *al.* [50] investigated ways to breach the isolation of MicroVM-based containers using operation forwarding attacks.

Our work draws inspiration from these prior studies on container security and provides an in-depth examination of the security challenges associated with the CoCo.

## 9 CONCLUSION

In this paper, we have undertaken a comprehensive assessment of Confidential Containers's attack surface and delved into the security vulnerabilities arising from the misalignment with the confidential computing principles. Our security experiments have revealed that by exploiting CoCo's unprotected control interface, adversaries can gain access to private data and manipulate the execution within the protected domains, effectively circumventing the protection of confidential computing. To bridge this security gap, we propose a redesign of CoCo's control interface, which involves constraining the capabilities of host-side controllers and granting workload owners the authority to manage their containers through an alternative secure tunnel. This approach ensures seamless integration with existing cloud-native orchestration layers and aligns CoCo with the threat model of confidential computing.

## ACKNOWLEDGEMENT

We extend our gratitude to Tobin Feldman-Fitzthum for his technical consultation and communication with the Confidential Containers community. Additionally, we sincerely thank our shepherd and the anonymous reviewers for their insightful comments and suggestions on this paper.

## REFERENCES

[1] 2022. Confidential Containers Architecture Overview. https://github.com/confidential-containers/documentation/blob/main/architecture.md.
[2] 2023. Release Notes for v0.8.0. https://github.com/confidential-containers/confidential-containers/blob/main/releases/v0.8.0.md.
[3] 2023. Security policy for Confidential Containers on Azure Kubernetes Service. https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-containers-aks-security-policy.
[4] AMD. 2023. AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/sev-tio-whitepaper.pdf. *White Paper* (2023).
[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 689–703.
[6] Thanh Bui. 2015. Analysis of Docker Security. arXiv:1501.02967
[7] CCv0. 2016. Kata Containers. https://github.com/kata-containers/kata-containers/commit/59d733f.
[8] Container Runtime Interface (CRI) CLI. 2019. https://github.com/kubernetes-sigs/cri-tools/blob/master/docs/crictl.md.
[9] Constellation. 2022. https://www.edgeless.systems/products/constellation/.
[10] Confidential Containers. 2023. https://github.com/confidential-containers.
[11] Enclave Confidential Containers. 2023. https://github.com/confidential-containers/enclave-cc.
[12] Kata Containers. 2023. https://katacontainers.io/.
[13] Gobikrishna Dhanuskodi, Sudeshna Guha, Vidhya Krishnan, Aruna Manjunatha, Rob Nertney, Michael O'Connor, and Phil Rogers. 2023. Creating the First Confidential GPUs. *Commun. ACM* 67, 1 (2023), 60–67.
[14] Christophe de Dinechin, David Gilbert, and James Bottomley. 2023. Attestation in confidential computing. https://www.redhat.com/en/blog/attestation-confidential-computing.
[15] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. 2017. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 237–248.
[16] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. 2019. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. 1073–1086.
[17] Xing Gao, Benjamin Steenkamer, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. 2018. A Study on the Security Implications of Information Leakages in Container Clouds. *IEEE Transactions on Dependable and Secure Computing* 18, 1 (2018), 174–191.
[18] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network and Distributed System Security (NDSS) Symposium*, Vol. 3. San Diega, CA, 191–206.
[19] Aaron Grattafiori. 2016. NCC Group Whitepaper: Understanding and Hardening Linux Containers.
[20] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. 2011. Process Implanting: A New Active Introspection Framework for Virtualization. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*. 147–156.
[21] Udit Gupta. 2015. Comparison between security majors in virtual machine and linux containers. arXiv:1507.07816
[22] Guerney D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakirman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. 2021. Confidential computing for OpenPOWER. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. 294–310.
[23] IBM. 2022. Introducing IBM Secure Execution for Linux 1.3.0. https://www.ibm.com/docs/en/linuxonibm/pdf/l130se03.pdf. (2022).
[24] Intel. 2023. Intel TDX Module 1.0 Specification. https://cdrdv2.intel.com/v1/dl/getContent/733568. (2023).
[25] Intel. 2023. Intel® TDX Connect Architecture Specification. https://cdrdv2.intel.com/v1/dl/getContent/773614. (2023).
[26] Intel. 2023. Intel® Trust Domain Extensions. https://cdrdv2.intel.com/v1/dl/getContent/690419. (2023).
[27] David Kaplan. 2017. Protecting VM Register State with Sev-es. *White paper* (2017).
[28] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD Memory Encryption. *White paper* (2016).
[29] Extend kubectl with plugins. 2021. https://kubernetes.io/docs/tasks/extend-kubectl/kubectl-plugins/.
[30] Kubernetes kubelet documentation. 2023. https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/.
[31] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. Speaker: Split-Phase Execution of Application Containers. In *Springer DIMVA*.
[32] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and Verification of the Arm Confidential Compute Architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 465–484.
[33] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. 418–429.
[34] Command line tool (kubectl). 2023. https://kubernetes.io/docs/reference/kubectl/.
[35] Yang Luo, Wu Luo, Xiaoning Sun, Qingni Shen, Anbang Ruan, and Zhonghai Wu. 2016. Whispers between the Containers: High-Capacity Covert Channel Attacks in Docker. In *IEEE Trustcom/BigDataSE/ISPA*.
[36] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.
[37] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. *The Second Workshop on Hardware and Architectural Support for Security and Privacy* 10, 1 (2013).
[38] Pradipta Banerjee. December 2, 2021. Restricting Kata Agent API. https://medium.com/kata-containers/restricting-kata-agent-api-e3dc88bf8270.
[39] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. GUITAR: Piecing Together Android app GUIs from Memory Images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 120–132.
[40] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2015. Vcr: App-agnostic Recovery of Photographic Evidence from Android Device Memory Images. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 146–157.
[41] Brendan Saltaformaggio, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. 2014. DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse. In *23rd USENIX Security Symposium (USENIX Security 14)*. 255–269.

[42] AMD SEV-SNP. 2020. Strengthening VM Isolation with Integrity Protection and More. *White Paper* (2020).

[43] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 955–970.

[44] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves.. In *Network and Distributed System Security (NDSS) Symposium*.

[45] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *27th USENIX Security Symposium (USENIX Security 18)*. 1423–1439.

[46] Tonic. 2022. A gRPC over HTTP/2 implementation focused on high performance, interoperability, and flexibility. https://crates.io/crates/tonic.

[47] Agent Control tool. 2022. https://github.com/kata-containers/kata-containers/tree/main/src/tools/agent-ctl.

[48] Trustee. 2023. Trusted Components for Attestation and Secret Management. https://github.com/confidential-containers/trustee/tree/main/kbs.

[49] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 645–658.

[50] Jietao Xiao, Nanzi Yang, Wenbo Shen, Jinku Li, Xin Guo, Zhiqiang Dong, Fei Xie, and Jianfeng Ma. 2023. Attacks are Forwarded: Breaking the Isolation of MicroVM-based Containers Through Operation Forwarding. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7517–7534.

[51] Yutian Yang, Wenbo Shen, Xun Xie, Kangjie Lu, Mingsen Wang, Tianyu Zhou, Chenggang Qin, Wang Yu, and Kui Ren. 2022. Making Memory Account Accountable: Analyzing and Detecting Memory Missing-account bugs for Container Platforms. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 869–880.