Bindox: An Efficient and Secure Cross-System IPC Mechanism for Multi-Platform Containers

Yuxin Xiang[1], Bing Deng[2], Hongyu Zhang[2], Randy Xu[2], Marc Mao[2], Yun Wang[1], Zhengwei Qi[1]
{xiangyuxin,yunwang94,qizhwei}@sjtu.edu.cn,{bing.deng,hongyu.zhang,randy.xu,marc.mao}@intel.com
[1] Shanghai Jiao Tong University, [2] Intel Corporation
Shanghai, China

Abstract

Containerization is widely used for isolation in various applications because it is lightweight, scalable, and portable. In modern distributed systems, seamless inter-process communication (IPC) between multi-platform containers is essential for a range of applications and services, including microservices, cloud computing, and Internet of Things (IoT) devices. However, secure and efficient communication between containers on the same host is challenging, especially when different operating systems are involved.

This paper introduces Bindox, a lightweight, efficient, and secure IPC mechanism that enables seamless communication across multiple platforms, including Android and Linux. Bindox uses shared memory for data transfer and implements a stable client-server architecture, ensuring high performance and ease of maintenance. Additionally, Bindox provides a robust security mechanism that guarantees confidentiality, integrity, and availability of the communication channel. Experimental results demonstrate that Bindox outperforms existing networking and IPC methods in terms of memory use, latency, and CPU usage, making it a promising solution for efficient and secure communication between multi-platform containers.

1 Introduction

Containerization has emerged as a transformative technology in modern computing due to its ability to provide an isolated runtime environment that is lightweight, scalable, and highly portable. In modern distributed systems, IPC between containers of different OS images has become increasingly important. Seamless communication between these multi-platform containers can enable a wide range of applications and services, such as microservices, cloud computing, and Internet of Things (IoT) devices [7].

However, fast and secure communication between containers on the same host can be challenging, particularly when different operating systems are involved. Traditional container communication methods on the same host can be classified into two categories: (a) networking, which includes using Docker networking and sharing the host network namespace; (b) IPC. which includes files, shared memory, UNIX sockets, pipes, semaphores, shared memory, and message passing. All these methods have their respective pros and cons (Section 2) and fail to strike a balance between performance and security. Inappropriate settings of both methods can introduce significant hazards to the system, which can be fatal when doing cross-OS interactions.

In recent years, several studies have attempted to address these challenges by introducing new IPC mechanisms [4, 6, 8, 16]. However, these mechanisms are often limited in terms of their compatibility with container environments or their ability to provide both high performance and security [15].

To address these challenges, in this paper, we introduce Bindox, a fast, secure, lightweight, and cross-system IPC mechanism that enables communication on and between multiple operating systems, including Android and Linux. Bindox leverages shared memory for data transfer to achieve high performance with zero-copy, which significantly reduces the data transfer overhead and improves performance. Bindox also provides a robust security mechanism that ensures the confidentiality, integrity, and availability of the communication channel.

To demonstrate the effectiveness of Bindox, we compare it to existing container-on-the-same-host communication methods, and the results show that Bindox shortens the transmission latency by 40% on average.

Bindox offers a promising solution for efficient, secure, and cross-platform communication between heterogeneous containers on the same host, which is critical for application performance, security, and cluster scalability. Compared to other solutions, Bindox has several advantages:

- Lightweight and flexible: Bindox is compatible with container environments and is suitable for use in various contexts and applications.
- High efficiency: Bindox achieves zero copy for data transfer, which reduces data transfer overhead and improves performance.
- Stable architecture: With a clear client-server architecture, Bindox simplifies the process of developing and maintaining the system. Unlike shared memory, users do not need to consider complex concurrency synchronization issues.
- Robust security: Bindox provides a robust security mechanism that ensures the confidentiality and integrity of communication between processes. This mechanism helps to prevent unauthorized access, data tampering, and other security threats.
- Cross-OS compatibility: Bindox supports cross-OS communication, including Android and Linux.

DOI reference number: 10.18293/SEKE2023-124

2 Background and Related Work

2.1 Cross-OS Communication

Communication between different operating systems is a ubiquitous requirement for modern applications. In fact, many applications critically depend on services hosted on the other OS to function properly. The seamless communication between such diverse services is essential for the development of robust and feature-rich applications.

The cloud platform is a typical example of cross-OS communication. With the emergence of various cloud-native technologies, such as Docker and Kubernetes, cloud computing have been widely adopted [12, 17], and the performance of IPC is crucial for achieving efficient cloud services. To extend the computing power of mobile devices and enable the cloud-based execution of Android applications, various approaches have been proposed for Android cloudification. For example, Android Emulator [2] and Cuttlefish [5] use virtual machines as isolation units to ensure that Android can run on any platform. To make the system lighter, Anbox [1] places Android into a container. Another solution, CARE [13], further cloudifies the Android system into a cloud-native system by streamlining Android services. In Android cloudification, Android applications are hosted on different operating systems, usually Linux. These VMs and containers rely on communication with the host to enable file sharing, network stack, and hardware device sharing. Moreover, the cloud Android applications can also utilize services on the host machine to function properly.

Autonomous car software serves as another example of Android-Linux communication. Typically, automotive software is divided into two main regions: automotive machine vision (AMV) and in-vehicle infotainment (IVI) [14]. The AMV region is responsible for the advanced driver assistance systems (ADAS) using a real-time Linux system [9], whereas the IVI region provides driving information and entertainment by utilizing an Android system [10] that is specifically tailored for consumer applications. Despite their distinct purpose, the two regions interact in various scenarios, sharing files, images, videos and sensor data on the same host. A common scenario is that an Android camera service transfers captured camera data to a Linux-based image analysis and processing module for vehicle and driver state analysis. Another example is that a real-time application running on Linux makes a decision, it needs to communicate to the media service running on Android, which can then notify the user through graphics or audio.

2.2 Container Communication On the Same Host

There are several methods for enabling communication between containers running on the same host, including networking and IPC methods. Networking methods encompass the use of Docker networking and sharing the host network namespace, while IPC methods include pipe, message passing, shared memory, UNIX sockets, semaphores, and signal. While these methods can provide fast and efficient communication, they fail to balance performance and security.

Networking methods allow containers to communicate directly with each other using IP addresses. However, it may cause considerable overhead due to the network protocol and stack, and it may pose security risks if not properly isolated from other networks.

While IPC methods, especially those that utilize shared memory, offer rapid and effective communication, their configuration and management can be intricate due to distinct namespaces. In addition, each IPC technique has restrictions, and inappropriate settings could result in substantial safety hazards, especially in cross-OS interactions. Thus, it is imperative to carefully consider the potential risks and carefully tailor the IPC method to the specific application and environment to ensure efficient and secure communication.

- Pipe: Pipes are half-duplex and limited to one-way communication between processes. To achieve bi-directional communication, two pipes are needed. Moreover, pipes are commonly used between parent and child processes and have a relatively small buffer size.
- Message queue: Message queues buffer size limits the amount of data to be transmitted. Moreover, message queues can lead to synchronization issues including deadlocks and priority inversion, which can negatively impact system performance.
- Shared memory: Shared memory is a high-performance IPC method that directly attaches shared buffers to a process's virtual address space. However, the synchronization between processes is left to the responsibility of the processes themselves. Furthermore, it may leak confidential data without proper access restriction.
- UNIX domain socket: Unlike network sockets, UNIX domain sockets are based on file system path names and do not require processing through the network protocol stack, thus offering higher performance and lower latency. However, UNIX domain sockets can introduce security risks if the socket is not properly secured. It is important to ensure that the socket is properly permissioned and that appropriate access controls are in place to restrict access. Moreover, they do not support more advanced features such as multicasting, which are often used in distributed systems.
- Semaphore: Semaphores are mainly used as lock mechanisms to prevent multiple processes from accessing shared resources simultaneously. Therefore, they are mainly used for inter-process and inter-thread synchronization.
- Signal: Signals are not suitable for information exchange but are instead useful for process interrupt control, such as handling illegal memory access or killing a process.

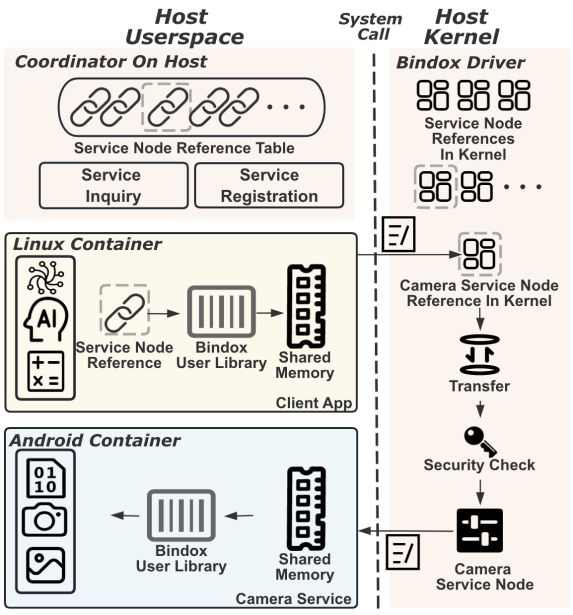


Figure 1. Bindox architecture

3 Bindox: Design and Implementation

3.1 Bindox Architecture

This section details the components of the Bindox design. Aiming at a user-friendly, secure-by-design, and cross-OS adaptive architecture, Bindox takes advantage of a stable client-server model to enable efficient and scalable IPC in a heterogeneous multi-OS container environment. As shown in Fig. 1, communications in Bindox involve four components: client, server, Coordinator, and Bindox Driver. In Bindox, clients and servers can be deployed on any host process or container. Coordinator is a daemon process running on the host, managing and serving all the clients and servers. Bindox Driver is implemented as a kernel module responsible for handling the low-level communication between different processes.

Client and server: Clients and servers are responsible for initiating and responding to communication requests, respectively. One server can handle clients in different platform containers, as shown in Fig. 2. Clients and servers can be implemented with the Bindox library, which is compatible with various operating systems, including Android and Linux. Upon each server's start, Driver creates a corresponding service node in the kernel space, which is the communication endpoint used to send messages between clients and services.

Coordinator: The Coordinator is a daemon process that serves as a central and secure registry for managing and accessing Bindox services. It maintains a table of available Bindox service references and provides service registration and discovery functionalities. The Coordinator also leverages the Bindox Driver to enforce security checks, ensuring that only authorized processes can access registered services.

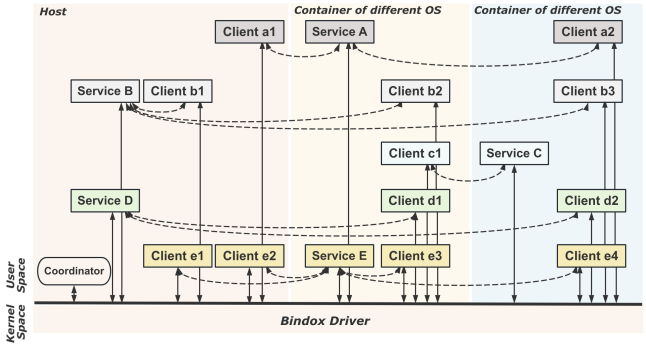


Figure 2. Bindox client-server communications

Coordinator makes it easier for developers to create complex and scalable applications that can communicate with other OS applications using Bindox.

Driver: The Driver is the core component of the entire communication system, responsible for creating and managing Bindox service nodes, handling the communication protocol, managing service references, and implementing security checks. The Driver is compatible with various OS kernels. It implements the low-level communication protocol, including sending and receiving message operations, and tracks and manages service references to services to automatically release resources when clients are no longer using them.

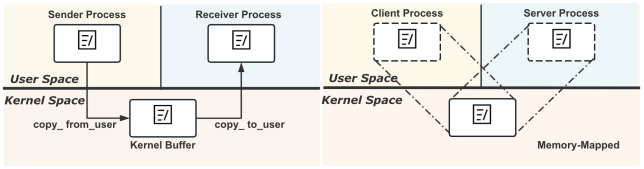
3.2 Bindox Communication Model

This section will explain the communication process in Bindox in detail. Given the scenario that on an autonomous vehicle, the advanced driver assistance system (ADAS) needs to access camera data, analyze, and process the images to determine whether the driver is experiencing driver fatigue. The host system is a real-time operating system, Automotive Grade Linux (AGL) [3], which is a collaborative open-source project used by multiple car manufacturers. The image processing program runs in a Linux container, while the camera image capture program runs in an Android container. As shown in Fig. 1, the image processing program acts as a client and requests image capture data from the camera service.

Coordinator Setup: Before using Bindox for communication, the Coordinator process must be started on the host. This process registers itself as Coordinator at the Driver and serves as a service registry and search center for further Bindox communications.

Service Registration: Upon starting the camera service, the server sends the service name and registration request to Coordinator through Driver. Driver creates the corresponding service node in the kernel and its reference, then forwards the service name and service reference to Coordinator. After receiving the data, Coordinator appends the name and service reference to the local table.

Service Discovery: The image processing application uses the service name to query access to the camera service



(a) Traditional IPC datapath **(b)** Bindox datapath

Figure 3. IPC and Bindox datapaths

from Coordinator via Driver. Driver first validates the request source, user capability, and requested data integrity, then forwards the request to the Coordinator. In service discovery, Coordinator's role is similar to Domain Name System (DNS), as it converts the requesting service name into service reference for clients. After Coordinator returns the service reference to the client, the camera service node now has two references: one in the Coordinator and one in the client. If more clients request the service in the future, the number of references to the camera service will increase accordingly.

Service Request and Respond: The image processing application can call the camera service through the service reference returned by Coordinator. Bindox Driver accomplishes all the low-level communication and security checks. Bindox security assurance includes user authentication, permission validation, and data integrity checking. However, all these details are hidden by Bindox Driver and Bindox user library, making Bindox concise and user-friendly.

3.3 Bindox Data Transfer

In addition to supporting cross-platform container communication, the most remarkable feature of Bindox is its high performance . Bindox achieves zero-copy during the transmission process, thereby minimizing the overhead associated with multiple copies.

In networking and conventional IPC, including pipes, message queues, and Unix sockets, at least two copies are needed during the message transfer. As depicted in Fig. 3a, these communication methods suffer from expensive copy operations, as at least two copies are required during message passing, which is particularly costly in frequent or high-volume communication scenarios.

To address these problems, Bindox proposes a new transfer method, *Mapping Delivery*. As shown in Fig. 3b, *Mapping Delivery* uses memory mapping to map the sender and the receiver user space buffers to the same physical memory, ensuring that data transfer from sender to receiver is only a logical copy without actual overhead. It is achieved through Bindox Driver. During communication, Bindox Driver provides a shared memory region that can be accessed by multiple processes, and establishes a mapping between memory, server's and client's user space addresses.

Through *Mapping Delivery*, Bindox can efficiently share memory between different processes in a secure and controlled manner. *Mapping Delivery* provides a shared memory region that can be accessed by multiple processes, and Bindox facilitates the communication and coordination between these processes. This combination allows for the creation of efficient IPC channels while maintaining a high level of security and access control. Specifically, *Mapping Delivery* provides a mechanism for allocating and sharing memory, while Bindox security design ensures that only authorized processes can access this memory and enforces strict security policies to prevent unauthorized access or tampering. This innovative approach enables the development of complex and highly performant systems while ensuring the security and integrity of shared data.

3.4 Bindox Security

The security mechanism of Bindox provides a robust defense against unauthorized access and helps to maintain the confidentiality, integrity, and availability of the system.

One of the innovative features of the Bindox security mechanism is the centralized Coordinator, which acts as a registry for all the available services in the system. Through Coordinator, Driver can checks the permission of clients asking for services. This enables the system to enforce fine-grained access control over the services and prevent unauthorized access or tampering.

Another innovative feature of the Bindox security mechanism is its use of permissions to control the interactions between components. It is based on the principle of least privilege, where every component of the system has only the minimum required permissions to perform its function. This is achieved through a series of access controls that limit the interactions between components. In Bindox, Bindox Driver mediates the interactions between processes and ensures that only authorized processes can access a particular service or component, and it prevents malicious components from causing harm to the system.

4 Evaluation

In this part, we present our evaluation of Bindox. We compare Bindox with networking and IPC methods, and try to answer the following questions:

- How does Bindox improve communication latency?
- What is the memory usage of Bindox during communications?
- What is the CPU usage of Bindox during communications?

4.1 Environment Setup

Configuration: All the experiments are conducted on a server running Ubuntu 22.04 LTS with Linux kernel 5.15, six

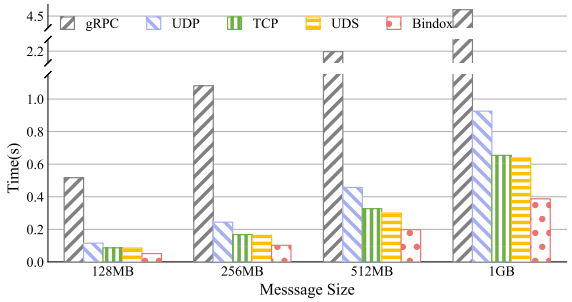


Figure 4. Average latency of message transfer iteration

Intel(R) Core(TM) i5-9400 CPUs at 2.90 GHz, and 15 GiB of physical memory.

Benchmarks: In the benchmark, the client process is held in a Docker Android container, and the server is running on the Linux host. Client and server alternately update the content of a fixed amount of memory and transfer the memory content to each other. Each iteration involves both client and server setting and passing memory once. To simulate real-world applications that often involve large-scale, precise, and secure data transfer, the memory size are set to 128MB, 256MB, 512MB, and 1GB. To ensure statistical accuracy, the iteration is set to 100 times during testing. This benchmark is designed to evaluate the efficiency and effectiveness of Bindox in handling data transfers.

Baselines: To illustrate the performance improvements brought by Bindox, we take networking and IPC methods as the baselines. In the networking baselines, the client container uses the host network. Three different network protocols are used: gRPC, TCP, and UDP*. gRPC is a modern open source Remote Procedure Call (RPC) framework that can run in any environment. It implements the same client-server architecture and security checks for communication as Bindox, but uses HTTP/2 as the underlying transport protocol. UDP* is a custom protocol built on top of UDP that guarantees transimission order and reliability, as we have found that in real-world scenarios, using UDP to transmit large amounts of data, even on the same host, often results in disorder and dropped packets. For IPC methods, we use UNIX domain socket as a baseline, as UNIX domain socket is widely used in container communication.

4.2 Transmission Latency

In this experiment, we present the average latency of message transfer iterations of Bindox.

Fig. 4 presents the time cost of different methods. We can witness that Bindox transfer latency is significantly lower than that of other communication methods. When transmitting data ranging from 128MB to 1G, the average transfer iteration time cost of Bindox is 40.7%, 39.2%, 39.5%, and 20.8% lower than that of TCP, the suboptimal communication method, respectively. Even with extra security checks,

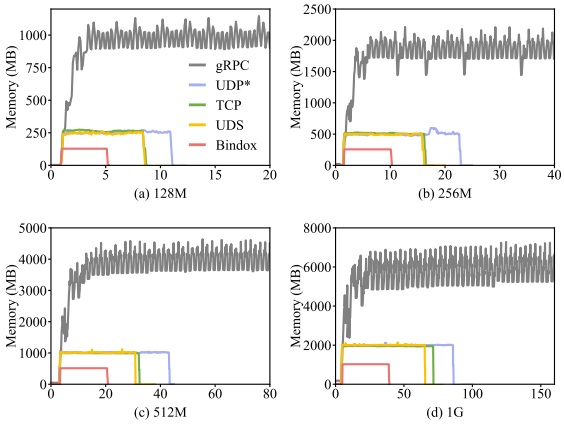


Figure 5. Total memory usage of client and server

Bindox has a significant performance advantage in handling reliable data transfer.

As Fig. 4 shows, gRPC has a much higher transmission time. This is because gRPC uses Protocol Buffers [11] as both its Interface Definition Language (IDL) and underlying message interchange format. Protocol Buffers types have language-specific implementations and require more data serialization and deserialization operations during transmission. Additionally, to ensure security, gRPC implements encryption transmission and authorization checks during function invocation, leading to a increased transmission time.

UDP* is based on the UDP protocol. It uses slicing and blocking to achieve reliable transmission, leading to a higher cost. The transmission cost of a UNIX domain socket is slightly lower than that of TCP. Although UNIX domain socket does not pass through the network stack, its bandwidth is lower than TCP. While UNIX domain socket has a significant performance advantage when transmitting small amounts of data (e.g., 1KB) compared to TCP, when transmitting large amounts of data, the time consumed by UNIX domain socket is comparable to that of TCP.

4.3 Memory usage

In this experiment, we measured the total memory usage of both client and server during transmission for each method.

As Fig. 5 presents, Bindox had a consistent memory usage pattern during message transmission. The memory consumption was stable and remained at about the size of the transmitted data. This is because Bindox uses a *Mapping Delivery* that avoids the need to copy data between processes.

In contrast, TCP, UDP, and UNIX domain socket all require data copying between the user space and kernel space, resulting in memory consumption that is twice the size of the transmitted data. As for gRPC, its memory consumption is observed to be highly variable during message transmission, and the average value is several times of the size of the transmitted data, as gRPC requires serialization and deserialization operations during data transmission, which increases memory usage.

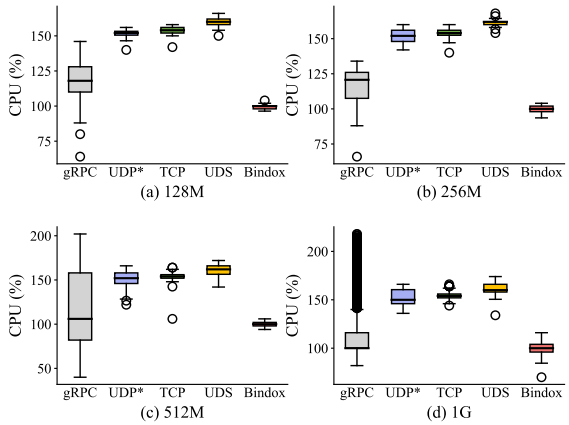


Figure 6. Total CPU usage of client and server

4.4 CPU usage

This experiment shows the total CPU usage of both client and server during transmission for each communication method.

As Fig. 6 shows, the total CPU usage of client and server of Bindox is relatively stable and lower than that of other methods. This can be attributed to the efficient design of Bindox *Mapping Delivery*, which utilizes shared memory to reduce the overhead of context switching and data serialization.

gRPC's is based on HTTP/2 protocol, leading to a higher CPU cost. However, its underlying message interchange format, Protocol Buffers, alleviates the CPU usage for message serialization and deserialization. For TCP and UDP*, these protocols rely on the operating system's network stack to process packets, resulting in frequent context switching and increased CPU overhead. Moreover, TCP and UDP* require additional overhead to handle packet fragmentation, retransmissions, and flow control. Additionally, TCP and UDP* perform checksum calculations on each packet, which also increases CPU usage. For Unix domain socket, it involves copying data between user and kernel space, and in the case of large data volumes, the CPU is polling for the kernel buffer to become available, resulting in significant CPU overhead.

5 Conclusion

Bindox offers a promising solution for efficient, secure, and cross-platform communication between heterogeneous containers on the same host. Compared to other solutions, Bindox is lightweight, flexible, and container-compatible, with a stable client-server architecture that simplifies development and maintenance. Additionally, Bindox provides a robust security mechanism that ensures the confidentiality and integrity of communication.

Acknowledgments

We thank for the reviewers' insightful feedback. This work was supported in part by National NSF of China (No. 62141218), Shanghai Key Laboratory of Scalable Computing and Systems, and Intel Corporation.

References

[1] Anbox 2023. *Anbox*. Retrieved February 27, 2023 from https://anbox.io/

[2] Android Emulator 2023. *Android Emulator*. Retrieved February 27, 2023 from https://developer.android.com/studio/run/emulator

[3] Automotive Grade Linux 2023. *Automotive Grade Linux*. Retrieved February 27, 2023 from https://www.automotivelinux.org/

[4] Susmit Bagchi. 2010. On Reliable Distributed IPC/RPC Design for Interactive Mobile Applications. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems* (Bangkok, Thailand) *(MEDES '10)*. Association for Computing Machinery, New York, NY, USA, 33–38. https://doi.org/10.1145/1936254.1936260

[5] CuttleFish 2023. *CuttleFish*. Retrieved February 27, 2023 from https://source.android.com/setup/create/cuttlefish

[6] D-Bus 2023. *D-Bus*. Retrieved February 27, 2023 from https://www.freedesktop.org/wiki/Software/dbus/

[7] Thomas Fischer, Christian Lesjak, Dominic Pirker, and Christian Steger. 2019. RPC Based Framework for Partitioning IoT Security Software for Trusted Execution Environments. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. 0430–0435. https://doi.org/10.1109/IEMCON.2019.8936247

[8] Zeyu Mi, Haoqi Zhuang, Binyu Zang, and Haibo Chen. 2022. General and Fast Inter-Process Communication via Bypassing Privileged Software. *IEEE Trans. Comput.* 71, 10 (2022), 2435–2448. https://doi.org/10.1109/TC.2021.3130751

[9] Milena Milosevic, Milan Z. Bjelica, Tomislav Maruna, and Nikola Teslic. 2018. Software Platform for Heterogeneous In-Vehicle Environments. *IEEE Transactions on Consumer Electronics* 64, 2 (2018), 213–221. https://doi.org/10.1109/TCE.2018.2844737

[10] Nemanja Pajic and Milan Bjelica. 2018. Integrating Android to Next Generation Vehicles. In *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*. 152–155. https://doi.org/10.1109/ZINC.2018.8448709

[11] Protocol Buffers 2023. *Protocol Buffers*. Retrieved February 27, 2023 from https://protobuf.dev

[12] Pengfei Shao and Shuyuan Jin. 2020. Privacy-aware OrLa Based Access Control Model in the Cloud. In *International Conference on Software Engineering and Knowledge Engineering*. Pittsburgh, USA, 216–221. https://doi.org/10.18293/SEKE2020-038

[13] Dongjie Tang, Cathy Bao, Yong Yao, Chao Xie, Qiming Shi, Marc Mao, Randy Xu, Linsheng Li, Mohammad R. Haghighat, Zhengwei Qi, and Haibing Guan. 2021. CARE: Cloudified Android OSes on the Cloud Rendering. In *Proceedings of the 29th ACM International Conference on Multimedia* (Virtual Event, China) *(MM '21)*. Association for Computing Machinery, New York, NY, USA, 4582–4590. https://doi.org/10.1145/3474085.3475617

[14] Srdjan Usorac and Bogdan Pavkovic. 2021. Linux container solution for running Android applications on an automotive platform. In *2021 Zooming Innovation in Consumer Technologies Conference (ZINC)*. 209–213. https://doi.org/10.1109/ZINC52049.2021.9499302

[15] Newton C. Will, Tiago Heinrich, Amanda B. Viescinski, and Carlos A. Maziero. 2021. Trusted Inter-Process Communication Using Hardware Enclaves. In *2021 IEEE International Systems Conference (SysCon)*. 1–7. https://doi.org/10.1109/SysCon48628.2021.9447066

[16] Zhoujian Yu, Cangzhou Yuan, Xin Wei, Yanhua Gao, and Lei Wang. 2016. Message-passing interprocess communication design in seL4. In *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*. 418–422. https://doi.org/10.1109/ICCSNT.2016.8070192

[17] Zekun Zhang, Bing Li, Jian Wang, and Yongqiang Liu. 2021. AAMR: Automated Anomalous Microservice Ranking in Cloud-Native Environment. In *International Conference on Software Engineering and Knowledge Engineering*. Pittsburgh, USA, 86–91. https://doi.org/10.18293/SEKE2021-091