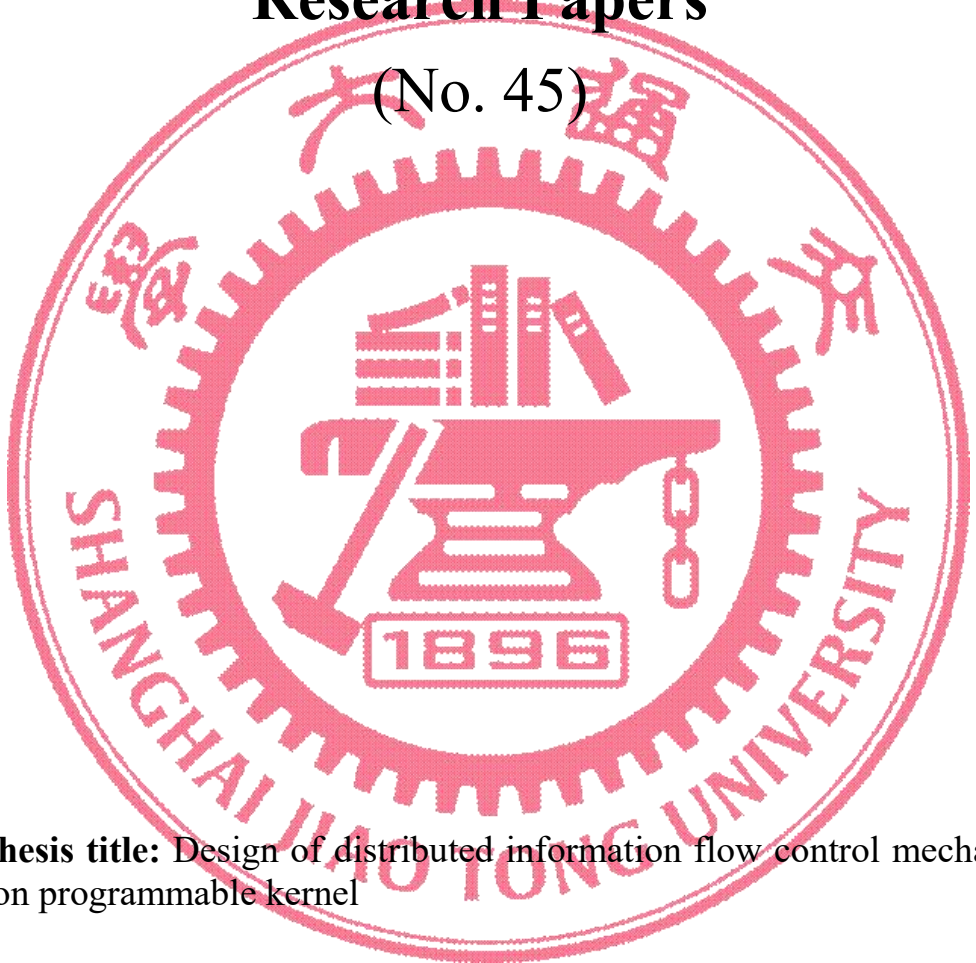
*[Translated by deepl.com]*
Item No. T030PRP45063

# Undergraduate Research Program (PRP) Research Papers

(No. 45)




**Thesis title:** Design of distributed information flow control mechanism based on programmable kernel

**Project leader**: Du Dongdong, School of Software

**Participating students**: Vahag Ghazaryan, Shi Zhan, Zhang Yue, Yiyu Luo.

**Project execution time**: February 2024 to September 2024

# Design of distributed information flow control mechanism based on programmable kernel


## ABSTRACT

This project focuses on designing and implementing a decentralized information flow control (DIFC) mechanism using programmable kernels (eBPF). As security becomes an increasingly critical priority for modern operating systems, DIFC emerges as a powerful approach for managing inter-process communication. The proposed mechanism extends existing security tools like bouheki by allowing the monitoring and blocking of specific system calls, such as signals, thereby enhancing system performance. This work provides valuable insights for the future development of security tools and establishes a foundation for applying this mechanism to the OpenHarmony operating system, addressing emerging kernel security challenges.


**Keywords: decentralized information flow control, programmable kernel, inter-process communication, eBPF, bouheki, pulsar**

## 1. Introduction

The increasing prominence of distributed systems and IoT devices has heightened the need for advanced security controls. OpenHarmony [1], led by the OpenAtom Foundation, aims to unify IoT device operations to enable seamless collaboration. Traditional static security rules are inadequate for modern complex distributed environments, particularly regarding security and data control.

Distributed Information Flow Control (DIFC) offers a flexible and effective security mechanism by allowing each node to independently manage and control information flow, reducing single-point-of-failure risks and providing fine-grained security management.

This study analyzes existing security tools like Pulsar, which use eBPF [2] for real-time monitoring but lack control over inter-process communication (IPC). To address this, we enhance Bouheki [3], a eBPF-based security auditing tool for Linux, by adding signal-level monitoring and blocking capabilities to manage IPC effectively. This extension enables Bouheki to intercept and block unauthorized inter-process signals, improving overall system security. We also plan to integrate DIFC into OpenHarmony to ensure secure data transmission between devices like smart homes, wearables, and automotive systems.

## 2. Background and related work

### 2.1 OpenHarmony operating system

OpenHarmony is an open-source operating system developed from Huawei's HarmonyOS and led by the OpenAtom Open Source Foundation [4]. It provides a unified OS solution for various IoT devices, enhancing seamless integration and promoting the connected device industry. As a distributed OS, OpenHarmony supports efficient hardware collaboration and resource sharing among multiple devices, ranging from low-power gadgets to complex terminals.

Enhancing OpenHarmony's security is crucial, as traditional mechanisms are insufficient for complex distributed environments. By introducing a programmable kernel mechanism, OpenHarmony combines core OS functionality with flexible control, allowing developers to customize flow control rules and ensure secure, orderly data transfer between devices.

### 2.2 What is Distributed Information Flow Control (DIFC)

DIFC is an advanced security technology designed to precisely manage and monitor information flow in distributed environments. As systems become more decentralized and dynamic, DIFC is vital for maintaining data confidentiality and integrity [5]. It moves away from traditional centralized models reliant on a single trusted control center, adopting a decentralized strategy where each node independently manages information flow. This enhances system robustness and security by reducing single-point-of-failure risks and allowing processes to set and enforce their own security policies.

Originally proposed by Myers and Liskov [6], DIFC supports component-level security policies, offering fine-grained management and secure data transmission between nodes. It plays a crucial role in various distributed applications, including cloud computing, IoT, big data processing, and microservices. For example, the Flume system applies DIFC at the OS process level, simplifying integration with existing applications and enabling secure interaction between legacy and DIFC-aware processes [5]. Continuous optimization of DIFC can lead to more secure and reliable distributed systems that protect user data while providing efficient services.

### 2.3 What is the Programmable Kernel Mechanism?

Early solutions like Kernel Modules (K-Modules) allowed dynamic extension of OS kernel functionality without rebooting. However, since K-Modules interact directly with the kernel, defects can cause instability or vulnerabilities.

To meet the need for secure and flexible kernel modifications, eBPF (Extended Berkeley Packet Filter) was developed. Initially designed for network packet filtering, eBPF has evolved into a powerful tool for extending Linux kernel functionality in a controlled manner. Unlike K-Modules, eBPF programs run in a sandboxed environment and undergo rigorous validation before loading, ensuring security and stability without direct kernel modifications.

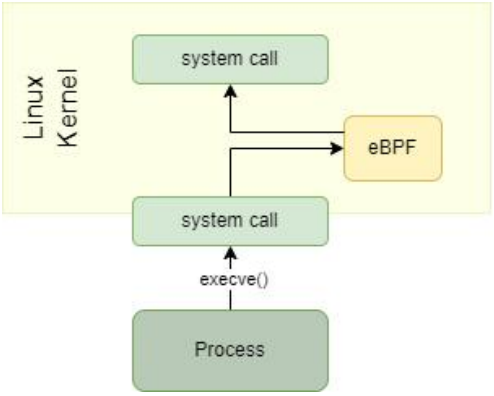



Figure 1 High-level structure of eBPF

eBPF enables developers to execute custom programs within the kernel to monitor and influence behavior in real time. These programs can access events like network packets, file operations, and system calls, allowing for detailed metrics collection, security rule enforcement, or real-time adjustment of kernel data handling without frequent user-kernel space switches, thus improving performance. Tools like bpftrace [7] utilize eBPF for in-depth system tracing, aiding in debugging and performance tuning. The main advantage of eBPF is its ability to dynamically and securely modify system behavior, enabling real-time functionality additions or observations without compromising system stability.

## 3. Analysis of existing methods

### 3.1 Overview of Pulsar Functions

Pulsar [8] is a security tool that utilizes eBPF to monitor Linux systems at runtime. It collects security-related events from the kernel, analyzes them to detect potential threats, and generates alerts based on predefined security rules. Pulsar enhances kernel security by providing real-time monitoring

and flexible threat detection capabilities. Key features of Pulsar include:

1. **System Resource Protection:** Restricts process privileges to prevent unauthorized access to sensitive files, storage, and network resources, maintaining system integrity.2. System integrity protection
2. **System Integrity Protection:** Monitors critical system events and configuration changes to detect unauthorized modifications promptly, preventing system instability.
3. **Network Security:** Monitors network activities such as tunneling and port forwarding to detect unauthorized remote access attempts, preventing data leakage.
4. **Limitations in Communication Control:** Pulsar lacks direct control over certain inter-process communication (IPC) mechanisms like pipes, shared memory, or domain sockets, limiting its ability to fully control inter-process data flow.

| System resources | | | | Communication | | | |
|---|---|---|---|---|---|---|---|
| FileStorage | Network | Process (permission) | Memory | Pipe | Shared memory | Socket (Domain) | Signal |
| v | v | v | v | x | x | x | x |

Table 1:Overview of Pulsar Functions

3.2 Pulsar Architecture and Design

Pulsar's architecture comprises two main components:

1. **Kernel-Mode Extensions**: Utilizes eBPF probes attached to key kernel events (e.g., file accesses, network operations) to collect system activity data in real-time with minimal overhead. These probes can immediately block harmful behavior by matching predefined conditions.
2. **User-Mode Agents**: Responsible for further analyzing the data collected by kernel probes. They perform in-depth evaluations based on complex security rules, analyzing patterns over time to detect anomalies.
3. **Data flows and interfaces:** Pulsar bridges kernel and user modes by exposing collected data through interfaces like the /proc filesystem. This allows user-space tools to access kernel-collected information in a structured format.
4. **Event Handling and Rule Evaluation:** eBPF probes capture events that are evaluated against security rules defined by administrators. By assessing these events in real-time, Pulsar can quickly generate alerts, enabling prompt threat mitigation.a powerful framework for effectively detecting and responding to security threats in real-time.

## 4. Content and methodology of the study

### 4.1 Overview

To address the limitations of existing tools like Pulsar, particularly in controlling IPC mechanisms, we aim to implement Distributed Information Flow Control (DIFC) mechanisms at the kernel level by extending Bouheki. By integrating signal-level monitoring and blocking using eBPF, Bouheki can effectively control signaling between processes, enhancing overall system security.

### 4.2 Signal

Inter-process signals are an important mechanism for inter-process communication in Linux systems, and are often used to deliver specific commands or alerts. However, they can also be used by malicious entities to perform unauthorized actions, such as terminating critical processes or disrupting system functionality. Controlling and monitoring signaling operations is essential for maintaining system stability and security. We have implemented signal-level monitoring and blocking functionality in Bouheki using eBPF programs attached to Linux Security Module (LSM) hooks (SEC("lsm/task_kill")) to intercept and manage signaling operations. With this mechanism, we can allow or block signaling between processes based on predefined policies, thus improving overall system security.

A YAML configuration file allows administrators to specify which signals should be blocked or allowed, applying restrictions based on criteria such as process ID, command name, or user ID. This provides fine-grained control over inter-process communication.

```
signals.
  mode: block # Setting mode to block specified signals
  type.
    deny: [2, 6, 9, 15] # Deny signals: SIGINT (2), SIGABRT (6), SIGKILL (9), SIGTERM (15)
    allow: ["*"] # Allow all other signals
  pid.
    allow: [1234] # Allow only specific process IDs
    deny: [] # Deny all others
  command.
    allow: ["safe_process"] # Allow specific command names
    deny: [] # Deny all others
  uid.
    allow: [] # No specific UID allowed
    deny: [] # Deny all others
```

This configuration allows fine-grained control of signals based on different attributes (such as PIDs and command names), resulting in more comprehensive management of inter-process communication.

The heart of the implementation lies in the eBPF program, which attaches to the task_kill LSM hook to intercept signaling operations. The following code snippet highlights the logic used to determine whether to block a signal.

```
SEC("lsm/task_kill")
int BPF_PROG(block_signal, struct task_struct *p, struct kernel_siginfo *info, int sig, const struct cred *cred) {
  ...
  // Check denied signals
  bpf_for_each_map_elem(&denied_types_signals, cb_check_path, &cb, 0);
  if (cb.found) {
    event.ret = -EPERM; // Deny signal
    ...
    return event.ret;
  }

  // Check allowed signals
  bpf_for_each_map_elem(&allowed_types_signals, cb_check_path, &cb, 0);
  if (cb.found) {
    event.ret = 0; // Permit signal
    ...
    return event.ret;
  }

  // Default block if no match found
  event.ret = -EPERM;
  ...
  return event.ret;
}
```

4.2.2 Test methods

The test consisted of configuring Bouheki using a YAML file to define policies for blocking specific signals, such as SIGTERM and SIGKILL. Bouheki was then run with this configuration and the blocked signals were used to locate background processes (e.g. sleep) via the kill command. The expected result is that these signals are intercepted and blocked, and an "operation not allowed" message is displayed, proving the effectiveness of our solution.

```
$ sleep 1000 &
$ kill -15 $!
 kill: sending signal 15 (SIGTERM) failed: Operation not permitted
```

## 6. Conclusion

This research successfully implements the DIFC concept and enhances Bouheki's ability to manage inter-process communication (IPC), including signals, thus improving system security in a distributed environment. By analyzing and addressing the limitations of existing tools (e.g., Pulsar), we extended Bouheki with eBPF-based monitoring and blocking capabilities, thus preventing unauthorized

communication and increasing system resilience. While full DIFC integration is still a work in progress, future work will focus on integrating these mechanisms into Bouheki and OpenHarmony to meet the need for flexible security controls in modern distributed systems.

## 7. References

[1] OpenHarmony, "OpenHarmony - Open Source Operating System," OpenAtom Foundation. Available: https://openharmony.io/

[2] eBPF Project, "What is eBPF," 2023. Available: https://ebpf.io/what-is-ebpf/

[3] Bouheki. (n.d.). Bouheki - KRSI (eBPF + LSM) Based Linux Security Auditing Tool. Available at:https://github.com/mrtc0/bouheki

[4] OpenHarmony Project, "OpenHarmony Overview," OpenAtom Foundation, 2023.Available: https://gitee.com/openharmony/docs/blob/master/en/ OpenHarmony-Overview.md

[5] . Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information Flow Control for Standard OS Abstractions," in Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP), MIT CSAIL & UCLA, 2007.

[6] . C. Myers and B. Liskov. "A Decentralized Model for Information Flow Control." in Proc. 16th SOSP, Oct. 1997.

[7] Bpftrace: high-level tracing language for Linux. (github.com)

[8] Pulsar. (n.d.). Pulsar - eBPF-based Security Tool. Available at: https://pulsar.sh/

## 8. Acknowledgements

First of all, I would like to thank my supervisor for his guidance and help throughout the research process. As an international student, I sometimes encountered language barriers in meetings and discussions, but my supervisor and fellow students in the group were always very patient in helping me by explaining and translating the parts that I did not understand so that I could better participate in the research. I would like to thank all the projects that make their research and source code publicly available, such as pular and bouheki, whose work has been an important support to my research.

This research experience was invaluable to me, not only was it my first in-depth involvement in scientific research, but it also inspired me to pursue a PhD after graduation to further explore this field. Therefore, I would also like to thank the organizers of the PRP program for giving me this opportunity to find my passion for research.