SACK: Enabling Environmental Situation-Aware Access Control for Vehicles in Linux Kernel

Boyan Chen[1,2], *Qingni Shen*[1,2*], *Lei Xue*[3], *Jiarui She*[1,2], *Xiaolei Zhang*[2],
Xiapu Luo[4], *Xin Zhang*[1,2], *Wei Chen*[1,2], *Zhonghai Wu*[1,2]
[1]School of Software and Microelectronic, Peking University
[2]National Engineering Research Center for Software Engineering, Peking University
[3]School of Cyber Science and Technology, Sun Yat-sen University
[4]Department of Computing, The Hong Kong Polytechnic University
[*]Corresponding Author

***Abstract*—Connected and autonomous vehicles (CAVs) operate in open and evolving environments, which require timely and adaptive permission restriction to address dynamic risks that arise from changes in environmental situations (hereinafter referred to as situations), such as emergency situations due to vehicle crashes. Enforcing situation-aware access control is an effective approach to support adaptive permission restriction. Current works mainly implement situation-aware access control in the permission framework and API monitoring in user space. They are vulnerable to being bypassed and are coarse-grained. Autonomous systems have widely adopted mandatory access control (MAC) to configure and enforce system-wide and fine-grained access control policies. However, the MAC supported by Linux security modules (LSM) relies on pre-defined security contexts (e.g., type) and relatively fixed permission transition conditions (e.g., exec syscall), which lacks consideration of environmental factors.**

To address these issues, we propose a Situation-aware Access Control framework in the Kernel (SACK), which enforces adaptive permission restriction based on environmental factors for CAVs. Incorporating environmental situations into the LSM framework is not straightforward. SACK introduces situation states as a new security context for abstracting environmental factors in the kernel. Subsequently, SACK utilizes a situation state machine to implement new adaptive permission transitions triggered by situation events. In addition, SACK provides a novel situation-aware policy language that links specific user space permissions to MAC rules while maintaining compatibility with other LSMs such as AppArmor. We develop two prototypes: an independent SACK with its own policies and a SACK-enhanced AppArmor that adaptively updates the corresponding policies of AppArmor. The experimental results demonstrate that SACK can efficiently enforce situation-adaptive permissions with negligible runtime overhead.

***Index Terms*—connected and autonomous vehicle, situation-aware access control, Linux security module**

I. Introduction

The software systems of connected and autonomous vehicles (CAVs) are inherently safety-critical and operate in evolving environments [18]. They gradually integrate safety-critical vehicle control functions (e.g., unlocking car doors) and perceive environment changes [6], [21]. Standards such as ISO/SAE 21434 [35] and AUTOSAR [16] underscore the necessity for environmental situation-aware security in CAVs. Environmental situations, henceforth referred to as **situations**, represent abstracted states of environments that can significantly impact the security and driving safety of CAVs [3], [18]. For example, the access control within in-vehicle infotainment (IVI) systems can not grant door-opening permissions to applications under normal situations (e.g., parking or normally driving) for security reasons while permitting such actions during emergencies (e.g., vehicle crashes) to ensure driver's safety [18]. In CVE-2023-6073 [2], an attacker can set the audio volume to its maximum in Volkswagen ID.3. It may threaten the driver's focus when the CAV is in a driving situation while it is not so dangerous in a parking situation. According to the principle of least privilege (POLP) in access control [30], subjects should only have necessary permissions in disparate situations. Therefore, supporting adaptive permission restriction by situation-aware access control is important for the security of CAVs.

In recent years, some research focus on situation-aware access control to follow the POLP. One approach [25], [27], [36] enforces situational policies within the permission framework to support situation-adaptive permissions in user space. Another approach [12], [22], [31] implements situation-aware security monitors at the API level to restrict permitted APIs in various situations. These approaches are all implemented in user space and offer clear policy expression and configuration, with easy access to semantic information about situations. However, some studies [8], [13], [14] demonstrate that attackers can easily bypass the permission framework and API checks in user space. After compromising or bypassing user space checks, many attacks [2] are performed by interacting with the kernel. In addition, permissions and API checks in user space are often too coarse-grained for the fine-grained control required in CAV hardware. Most existing works [22], [36] track situations when performing access control decisions, thus entangling access control enforcement with situation tracking, leading to overprivileged, inconsistent, and inflexible implementations [31].

Introducing situations into kernel space access control can mitigate the above issues. Most CAV software systems, such as IVI OSes [21], are based on the Linux kernel, and they implement mandatory access control (MAC) that is supported by the Linux security module (LSM) framework. Our insight is that situation awareness is critical for MAC regarding the following advantages. First, by implementing situation-aware access control directly within the Linux kernel, we can effectively prevent attacks that bypass user-space security monitors or exploit system calls. Second, situation-aware MAC

978-3-9826741-0-0/DATE25/© 2025 EDAA

offers fine-grained and direct control over CAV hardware. Finally, by decoupling situation tracking (in user space) from access control enforcement (in the kernel), we achieve better consistency and flexibility while adhering to the POLP.

In this work, we propose a Situation-aware Access Control framework in the Linux Kernel (SACK) to achieve situation-aware adaptive permission restriction for CAV software systems. However, the LSM framework was originally designed for relatively static environments like the PC and server [37], and the permission transition conditions (e.g., exec syscall) are also fixed and static. So, we encounter three major research challenges. **C1** is how to represent complex situations and transmit them to the kernel securely with low latency. **C2** is how to bridge the gap between user and kernel space permissions with situation-aware policy language. **C3** is how to achieve situation-aware adaptive policy enforcement with low runtime overhead.

To address **C1**, we propose the "situation state" as a new security context to separate situation tracing and access control enforcement. SACK monitors situation events that trigger situation state transitions and only transmits situation events when detected. SACK employs a securityfs-based approach to transmit situation events, which has security and integrity guarantees from the LSM framework. To address **C2**, we design a user-friendly policy and unified interface to specify situation states and corresponding "State-Permission-MAC rules" mapping, which separates policy and implementation to be compatible with other security modules. This mapping mechanism bridges the gap between user space permissions and MAC rules. To address **C3**, we establish a new situation-aware permission enforcement mechanism supported by a situation state machine to maintain the current situation state. We also elaborately design the situation state transition and permission mapping algorithm.

The main contributions of this paper are given as follows:

- We propose SACK, a situation-aware access control framework in the kernel for CAVs. Our approach provides fine-grained adaptive permission restriction based on the situations while effectively addressing circumvention and coarse-grained problems in previous user-space approaches.
- We first introduce the situation state as a new security context in the LSM framework to enable a dynamic and situation-aware permission transition in the kernel. Then, we also design a general high-level situation-aware policy language that introduces a permission mapping mechanism to establish correlations between user space permissions and MAC rules. Finally, we propose an adaptive permission enforcement mechanism in the kernel involving a situation state machine driven by situation events.
- We implement a prototype of SACK and extensively evaluate it. The results demonstrate that SACK can achieve situation-aware access control for CAVs with negligible runtime overhead (average below 3%).

II. Related Work and Motivation

A. Related Work

1) Situation-aware access control for smartphone: Explicit situation-dependent constraints for Android have been proposed in Apex [25], CRePE [9], MOSES [42], FlaskDroid [7] and Shebaro et al. [32]. Policy rules are activated and deactivated dynamically by configurable detectors (e.g., "context providers" in FlaskDroid [7] and Shebaro et al. [32]). These approaches all assume that situation tracking is done on the Android platform and using its capabilities. Later, machine learning has been used to automate situation-aware permission decision-making [15], [27], [36], but they have a relatively high false positive rate and are not reliable enough for CAV.

2) Situation-aware access control for smart home: Most of the existing IoT framework [12], [31], [43] depend on environmental situations in access control policies. Zhou et al. [43] propose a framework to discover the safety issues of interactions between devices and the real world. IoTSAFE [12] enforces safety and security policies with real-world IoT physical interaction discovery at the API level. Schuster et al. [31] propose environmental situation oracles in the IoT API layer to enable flexible situation-aware access control and follow the POLP. Optimistic access control (OAC) operates by limiting critical permissions by default but allows for exceptional access to "break the glass" in emergencies, which has been proven by Malkin et al. [23] that it is suitable for the smart home to balance safety and usability. We believe that OAC also applies to CAVs because of the same safety-critical nature.

3) Situation-aware access control for CAVs: Researchers also made a preliminary attempt at situation-aware access control in the vehicle. Gansel et al. [17] enforce context-based access control for car screen pixels. There are some in-vehicle access control frameworks that take situations into consideration. Gupta et al. [19] implement attribute-based access control (ABAC) for the in-vehicle network where attributes can be location and longitude. AVGuardian [20] proposes the publish-subscribe over-privilege problem in autonomous vehicle software systems such as Baidu Apollo, and it mitigates this problem by designing a run-time publish/subscribe permission policy enforcement mechanism to perform online policy violation detection and prevention. [18] summaries context-aware security and safety issues of CAVs.

4) Linux security module: The LSM framework was introduced in 2002 [37], which provides an interface that allows Linux to implement MAC. Most CAV systems also implement MAC. For example, the Tesla Version utilizes AppArmor [1], [26] to enforce MAC policies. Most security modules are based on the type enforcement (TE) model [5] where access decisions are based on the types of subjects and objects. Policies are written according to these types. Most well-known security modules, such as AppArmor, define and enforce their security policies at load time. AppArmor operates using a comprehensive policy loaded at boot time and typically does not change dynamically [1]. It does not inherently consider environmental factors, such as driving or parking. Few works partly introduce environmental information into kernel space access control. Varshith et al. [33], [34] implement ABAC in the LSM framework. They contain only limited environmental attributes, such as time or day of the week, and are not compatible with other security modules such as AppArmor.

B. Motivation

Existing situation-aware access control works [7], [12], [17], [20], [27], [31] mainly achieve situation-aware security monitors in user space. However, various attacks can bypass the user space approaches [21]. For example, KOFFEE (CVE-2020-8539) [11] allows an attacker to bypass security monitors in user space and inject vehicle control commands. Once attackers bypass situation-aware access control in the user space, they can utilize dispensable permission to execute malicious functions even with the protection of the LSM framework. Moreover, compared with MAC, permissions and API checks in user space are often too coarse-grained for controlling CAV hardware. Most works regard situation tracking as an integral component of access control implementation, resulting in issues related to redundant privileges and inflexibility [31].

Implementing situation-aware access control in the kernel mitigates attacks that circumvent user-space approaches. Because MAC provides a more comprehensive, enforceable, and scalable security model compared with user-space access control. In addition, this approach provides fine-grained and direct control of CAV hardware. Lastly, by separating situation tracking in user space from enforcement in the kernel, we achieve improved consistency and flexibility while adhering to the POLP and OAC. For example, Granting door-open permissions in normal situations poses a security risk because it violates the POLP and attackers could leverage this dispensable permission. However, once the vehicle crashes and the driver loses consciousness, timely evacuation or rescue becomes critical. By applying the principles of OAC, system services can be granted the necessary permissions such as windows or door control permissions (e.g., `CONTROL_CAR_DOORS`) in the kernel in emergencies. This allows the system to automatically open windows and unlock doors, facilitating quick passenger evacuation or enabling rescuers to access the vehicle's interior.

III. SACK Design

A. Threat Model

The security assumptions of this paper are similar to security modules such as AppArmor [1] and other MAC approaches [7], [29]. Attackers can not get root privilege in the kernel or get `CAP_MAC_OVERRIDE` capability and can not modify MAC policies. Additionally, similar to prior situation-aware access control approaches [7], [22], [31], we assume that the environmental information perception is trusted. We consider attackers from the application and middleware layer in user space, including over-privileged apps and libraries. The attacker can launch the attack through either (i) malware, in which the malicious logic is embedded at the install time, or (ii) vulnerable apps, which contain design or implementation flaws that can be exploited by a co-located malicious app or a remote network attacker to escalate its permissions and cause damages such as unauthorized device control. We also assume attackers can bypass the permission framework and API checks, so our trusted computing base (TCB) is smaller than user space approaches. Notably, attacks targeting vulnerabilities of the Linux kernel or breaching the integrity of security policies are out of the scope.

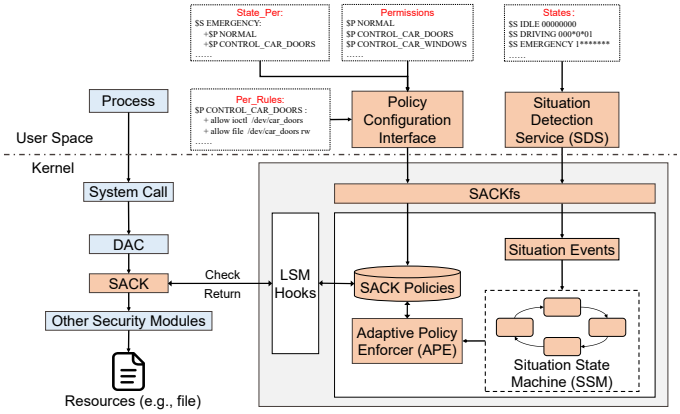


Fig. 1: SACK architecture.

TABLE I: Policy interface description of SACK.

Policy interface	Description
States	Specify situation states and their encodings
Permissions	Define permissions in SACK
State_Per	Specify "State-Permission" mapping
Per_Rules	Specify "Permission-MAC rules" mapping

B. SACK Architecture

In SACK, our core idea is to design and implement the situation state as a "security administrator" in the kernel, which further dynamically enforces MAC policies corresponding to the current situation state. As Fig. 1 shows, SACK embraces a two-layer design that includes a situation detection service (SDS) that monitors environment information (e.g., location, speed) and detects situation events (e.g., vehicle crashes occurred) in user space to achieve situation awareness and transmission. SACK utilizes a situation state machine (SSM) with the adaptive policy enforcer (APE) in kernel space to achieve situation-aware policy enforcement.

C. Situation Awareness and Transmission

The environment is complex and constantly changing. To address **C1**, SACK needs to detect situation changes in the kernel efficiently and with low latency. We introduce the situation state as a new security context in the kernel. Situation states separate situation tracing and access control enforcement, which bridges the gap of situation awareness between user and kernel space. SDS monitors situation events and only transmits them when detected. Situation event transmission requires low latency and security. Socket or system call-based approaches cannot satisfy these requirements. Thus, we use "SACKfs" to transmit situation events, which is based on securityfs [10] with security, integrity, and efficiency guarantees in the LSM framework.

D. Policy Language and Configuration in SACK

The permission framework is easy to understand but too coarse-grained. MAC rules are fine-grained but huge and complex, so configuring MAC policies is a laborious task. To address **C2**, SACK aims to bridge the gap between them and provide a user-friendly and unified policy interface. Table I shows the above policy configuration interfaces of SACK and

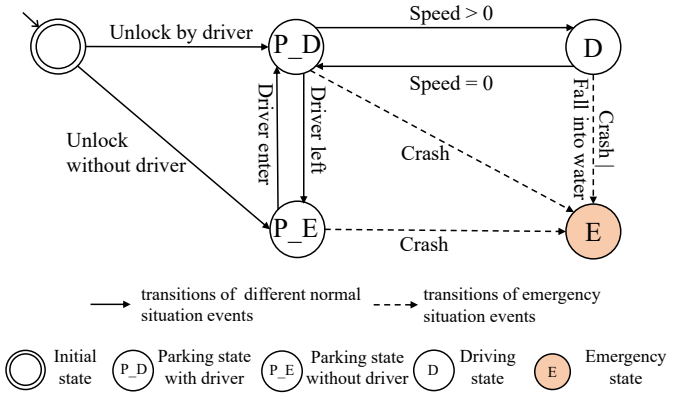


Fig. 2: Situation state machine example in SACK.

its functions. SACK provides four policy interfaces to specify situation states (States interface), permissions (Permission interface), and the corresponding "State-Permission-MAC rules" mapping (State_Per and Per_Rules interface). Fig. 1 depicts corresponding example policy files of SACK, the Sate_Per specifies that the emergency state contains `NORMAL` and `CONTROL_CAR_DOORS` permissions, and Per_Rules specifies that the MAC rules correspond to `CONTROL_CAR_DOORS` permission. SACK separates policy and implementation to ensure compatibility with different enforcement approaches. Our policy-checking tools also handle errors and conflicts. These interfaces let security administrators specify the situation states, permissions, state-permission mapping, and permission-MAC rules mapping. Like most Linux security modules, SACK also utilizes securityfs [10] to configure these policies, which provides security, integrity, and compatibility.

E. Situation-Aware Permission Enforcement

To address **C3**, we design SSM to maintain the current situation state, which is responsible for state transition based on situation events. The SSM provides a new permission transition condition triggered by situation events in the kernel. Unlike permission transition based on different labels of subjects and objects, SACK also builds new unified and complementary situation-aware permission transition conditions. Furthermore, SACK enforces adaptive permissions in two approaches. SACK can utilize its policies to perform permission checks. Another approach is integrating SACK with other security modules, such as AppArmor. Thus, SACK only updates AppArmor's policies when the current situation changes.

1) Situation state machine building: SACK builds SSM to represent situation states, transition rules, and situation events. Upon a situation event is detected by the SDS, it forwards the situation event to the SSM via securityfs-based transmission interface. The SSM then verifies if this event matches any of the transition rules associated with the current situation state, facilitating a transition from the current state to a target state when there is a match. Fig. 2 shows a SSM example with 4 situation states: emergency, driving, parking with driver, and parking without driver states. Emergency situation events such as vehicle crashes trigger the situation state transition from normal to emergency. After the SSM is built and runs in the kernel, it captures situation events from the SDS and determines whether it transits from the current state. Furthermore, SACK needs to map and enforce the corresponding permissions dynamically according to the current situation state.

Algorithm 1: *Situation state transition and policy mapping*

Input: Finite state machine **SSM** $= (SS_{0-n}, SE_i, TR_i, q_0, F)$ where SS_{0-n}, SE_i, TR_i represents situation state, situation event, transition rules respectively; q_0 and F is initial and finite state
SACK permission mapping function: $P_j = f(SS_i)$
MAC rule mapping function: $MR_k = g(P_j)$.
Output: Current MAC rules: $MR_{current}$.
$SS_{current} = q_0$
while $true$ **do**
  if $SE_{current} \neq NULL$ **and** $(SE_{current}, SS_{current})$ match TR_i **then**
    $SS_{current} = TR_i(SE_{current}, SS_{current})$
    $P_{current} = f(SS_{current})$
    $MR_{current} = g(P_{current})$
  end if
end while

2) SACK permissions and policy mapping: MAC provides fine-grained control over resources by different labels of subjects and objects. But MAC rules are complex, extensive, and difficult to comprehend [41]. Therefore, directly incorporating MAC rules into SACK policies is not practical. Conversely, the user space permission framework is relatively easy to comprehend, but it is not originally designed to restrict kernel-level system resources and is too coarse-grained. Bridging the gap between SACK permissions and access control policies is challenging.

Situation state transition and policy mapping algorithm. A situation state is explicitly associated with the allowed SACK permissions. When the state of SSM transitions, the SACK permissions (P_i) corresponding to the current situation state ($SS_{current}$) are also mapped by predefined policies. Each policy is a triple: (SS_i, P_i, MR_i), where P_i is the allowed SACK permissions corresponding to situation state SS_i. MR_i is the MAC rules corresponding to P_i. The permission and MAC rules mapping are also described in Algorithm 1. The APE maps the access control policies corresponding to $SS_{current}$. This approach is easy to adopt in the Linux kernel and maintains compatibility with existing LSMs.

3) Runtime adaptive policy enforcement: MAC policies are typically configured beforehand and seldom updated after deployment. Therefore, how to enforce permissions at runtime is still challenging. We show that the SACK mechanism can be implemented as ··independent SACK" with its own access control policies. This approach has the advantage of enforcing policies with low latency because SACK queries its own access control policies to return decisions. Independent SACK can not leverage the current MAC policies of existing LSMs. We also show that SACK can be implemented as "SACK-enhance AppArmor", where SACK updates and enforces the policies of

AppArmor when the current situation state transitions.

IV. Evaluation

A. Experimental Setup

Implementation. We implement independent SACK and SACK-enhanced AppArmor with Linux Kernel 5.10.0. In both ways, SACK is an independent security module. Independent SACK contains a total of 1700 LOC (lines of code), with the implementation of the security model accounting for 1000 lines C code. The user space programs account for 400 LOC. The remaining parts account for 300 LOC. SACK-enhanced AppArmor contains 650 lines C code in the kernel. The other parts are basically the same as independent SACK. We evaluate different SACK implementation approaches.

Evaluation Design. We extensively evaluate SACK to answer the following questions corresponding to design goals.

- **Q1:** How about the run-time performance of SACK?
- **Q2:** How about the security and safety enhancement of SACK?
- **Q3:** How about the compatibility of SACK with other Linux security modules such as AppArmor?

Evaluation platform. We test SACK on a PC platform with Intel I7-14700K, 32GB 6000HZ DDR5 memory, and 2TB SSD. We also deploy SACK in Starfive Visionfive2 [4] RISC-V single board embedded computer, which is also adaptive to next-generation CAV systems. We also partly evaluate the runtime overhead and perform a case study on this RISC-V board.

B. Performance

1) Runtime overhead: To answer **Q1**, we utilize LMBench [24] to evaluate the runtime performance of SACK. LMBench is a micro-benchmark that tests the performance of critical operations in OS. As Table II shows, we chose four kinds of operations to compare AppArmor with two implementations of SACK, all by default policies, and it reflects the overhead introduced by SACK in existing LSMs such as AppArmor. The experimental result shows that SACK can act as an extension of AppArmor and only cause negligible additional overhead. The permission check process for SACK-enhanced AppArmor is the same as that for the original AppArmor. Furthermore, we evaluate the performance of independent SACK on a RISC-V board utilizing LMBench, and the results indicated that in comparison to the original system without LSM framework (we need to enable LSM for SACK, and it also incurs overhead), SACK led to an average increase of 4.53% in file read and 6.36% in file write overhead.

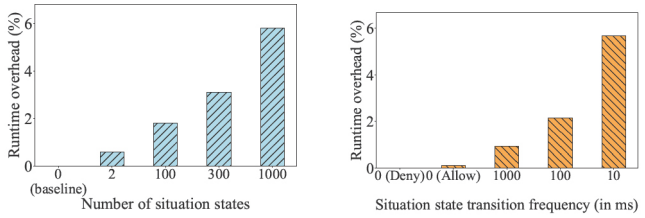
Overhead with the different number of rules. SACK policies are composed of many situation policies. As the number of policies grows, the overhead will grow accordingly. We utilize LMBench to evaluate the overhead of SACK with a multiple number of rules from four aspects, and we test 30 times in each case. The test policies are shown in Fig. 1. As Table III shows, the number of SACK rules causes neglectable run-time overhead for OS. Many of the data differences can be attributed to errors and jitter. Part of the reason is that the number of policies has little effect on the process of MAC check in AppArmor.

TABLE II: LMBench result of SACK.

Configurations	AppArmor (baseline)	SACK-enhanced AppArmor	Independent SACK
Processes (times in ms - smaller is better)			
syscall	0.039	0.040 (↓2.56%)	0.040 (↓2.56%)
fork	44.758	45.092 (↓0.75%)	41.443 (↑7.41%)
stat	0.198	0.199 (↓0.51%)	0.199 (↓0.51%)
open/close file	0.505	0.503 (↑0.40%)	0.517 (↓2.38%)
exec	115.391	114.550 (↑0.73%)	117.321 (↓1.67%)
File Access (in ms - smaller is better)			
file create (0K)	2.733	2.725 (↑0.29%)	2.737 (↓0.15%)
file delete (0K)	1.923	1.864 (↑3.07%)	1.918 (↑0.26%)
file create (10K)	5.575	5.573 (↑0.04%)	5.579 (↓0.07%)
file delete (10K)	3.113	3.081 (↑1.03%)	3.136 (↓0.74%)
mmap latency	7116.670	7161.690 (↓0.63%)	7189.232 (↓1.02%)
Local Communication Bandwidths (in MB/s - bigger is better)			
pipe	5698.170	5710.760 (↑0.22%)	5682.120 (↓0.28%)
AF_UNIX	11.720K	11.600K (↓1.02%)	11.820K (↑0.85%)
TCP	14.070K	14.240K (↑1.21%)	14.160k (↑0.64%)
File reread	13.490K	13.476K (↓0.10%)	13.501K (↑0.08%)
Mmap reread	24.450K	25.682K (↑5.04%)	25.123K (↑2.75%)
Context Switching (in ms - smaller is better)			
2p/0K ctxsw	1.234	1.232 (↑0.16%)	1.235 (↓0.08%)
2p/16K ctxsw	1.259	1.290 (↓2.46%)	1.267 (↓0.64%)

TABLE III: LMBench result of the different number of rules in AppArmor with SACK.

Policy numbers	0 (baseline)	10	100	500	1000
Processes (times in ms - smaller is better)					
syscall	0.04	0.04 (0%)	0.04 (0%)	0.04 (0%)	0.04 (0%)
I/O	0.06	0.06 (0%)	0.06 (0%)	0.06 (0%)	0.06 (0%)
File Access (in ms - smaller is better)					
file create (0K)	2.75	2.72 (↑1.09%)	2.73 (↑0.73%)	2.72 (↑1.09%)	2.72 (↑1.09%)
file delete (0K)	1.92	1.90 (↑1.04%)	1.91 (↑0.52%)	1.89 (↑1.56%)	1.89 (↑1.56%)
file create (10K)	5.61	5.62 (↓0.18%)	5.60 (↑0.53%)	5.60 (↑0.18%)	5.60 (↑0.18%)
file delete (10K)	3.12	3.17 (↓1.60%)	3.08 (↑1.28%)	3.10 (↑0.64%)	3.10 (↑0.64%)
mmap latency	7119.43	7122.63 (↓0.04%)	7138.53 (↓0.27%)	7131.50 (↓0.17%)	7128.60 (↓0.13%)
Local Communication Bandwidths (in MB/s - bigger is better)					
pipe	5627.43	5598.80 (↓0.51%)	5575.47 (↓0.92%)	5574.83 (↓0.93%)	5527.80 (↓1.77%)
AF_UNIX	11.63K	11.73K (↑0.86%)	11.67K (↑0.34%)	11.80K (↑1.46%)	11.60K (↓0.26%)
TCP	14.23K	13.87K (↓2.57%)	14.43K (↑1.41%)	14.30K (↑0.47%)	14.27K (↑0.28%)
File reread	13.35	13.44K (↑0.70%)	13.53K (↑1.34%)	13.44K (↑0.67%)	13.49K (↑1.05%)
Mmap reread	24.82K	26.49K (↑6.73%)	23.85K (↓3.91%)	25.30K (↑1.93%)	24.76K (↓0.24%)
Context Switching (in ms - smaller is better)					
2p/0K ctxsw	1.17	1.23 (↓4.88%)	1.23 (↓5.13%)	1.22 (↓4.29%)	1.22 (↓4.29%)
2p/16K ctxsw	1.21	1.29 (↓6.20%)	1.29 (↓6.61%)	1.29 (↓6.78%)	1.29 (↓6.61%)



(a) Runtime overhead with different number of situation states.

(b) Runtime overhead with different state transition frequency

Fig. 3: Runtime overhead of different situation states and state transition frequency.

Overhead with different number of situation states. SACK policies may have a varying number of situation states. Therefore, we define different test situation states and evaluate independent SACK, which is a worst-case scenario compared to SACK-enhanced AppArmor because SACK implements its LSM hooks. Fig. 3 (a) shows the runtime performance with multiple situation states. The result shows SACK brings 1.8% runtime overhead for file operations in 100 states.

Overhead with different situation state transition frequencies. The transition frequency between situation states also

results in varying levels of runtime overhead. We test the overhead at millisecond granularity transition frequency. We set two situations: high-speed and low-speed situations. Access to a critical file is only allowed in low-speed situations. We measure the overhead of transitioning between the two situations at multiple frequencies. Fig. 3 (b) shows that the results indicate that performance overhead is only 0.93% at a frequency of 1000ms.

Situation awareness latency. We evaluate the securityfs-based user/kernel space situation awareness latency with four situation events. The experimental results show that the average latency is around 5.4μs with 100% accuracy.

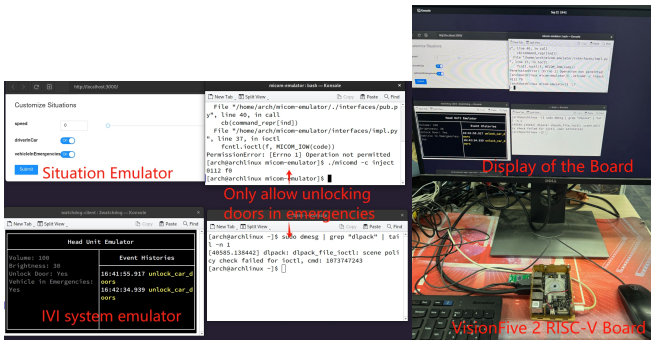
C. Security and safety enhancement brought by SACK

To answer **Q2**, we deploy a simulation environment to apply a case study to show the security and safety enhancement. We also analyze the security of SACK itself.

Simulation methodology. It is impractical for us to replace or update the kernel of the existing CAV software systems in real vehicles. Primarily due to the operational, safety, cost, and closed-source characteristics embedded within automotive systems, we can not trigger emergency events such as a vehicle crash on a real vehicle. KOFFEE [11] provides a tool to simulate IVI systems and this attack. Based on this tool, we develop an IVI emulator and execute command injection attacks. Furthermore, we evaluate SACK on a Visonfive2 RISC-V board [4] and PC platform, then emulate different events by writing a pseudo-file interface (located at */sys/kernel/security/SACK/events*).

1) Case study - allow unlock car door only in emergencies: The `CONTROL_CAR_DOORS` permission is extremely sensitive and should not be granted to the process in normal situations. However, when a vehicle accident occurs, timely opening of the car doors is important for passengers to escape and for rescuers to rescue. Some commercial CAVs provide vehicle crash detection functions [28]. The privileged rescue daemon can control car doors in such emergencies. As Fig. 4 shows, We test this case in Visonfive2 RISC-V board. SACK defines normal and emergency situation states in this case study and specifies the following SACK policy: In normal situation state, ioctl and write operations are not allowed to be executed on the window and door devices. But in the emergency situation, it is allowed to send window and door operation requests by executing the specific ioctl system call. Then, a react app triggers the vehicle crash event. During this emergency, the rescue daemon can control car doors and windows. In this experiment, control of windows and doors through a specific ioctl system call is allowed only when emergency events occur.

2) Security analysis of SACK: We analyze the security of SACK from different aspects. First, the security protection level of kernel space components of SACK is the same as that of LSMs such as SELinux and AppArmor because SACK is implemented as an independent LSM module. Being part of the kernel means that SACK operates with the highest privileges and is protected by the same mechanisms that protect the kernel, such as capability (e.g., `CAP_MAC_ADMIN`, `CAP_MAC_OVERRIDE`). Second, SACK utilizes securityfs to



(a) Display in this case study. (b) Our test hardware environment.

Fig. 4: Case Study: only allow unlock car door only in emergencies.

transit situation events. securityfs [10] is a filesystem provided for security modules; it thus looks from user space, like part of sysfs, but it is a distinct entity. securityfs is also protected by other security mechanisms, such as capability. Moreover, securityfs also provides security and integrity features for the SACK policy. Lastly, the LOC of SACK is relatively small.

D. Compatibility

Evaluation of the compatibility with AppArmor. We highlight that SACK provides compatibility with existing Linux security modules such as AppArmor. SACK leverages LSM stacking to ensure compatibility with other security modules such as AppArmor. In LSM stacking, the checking order follows a white-list based approach. SACK is set as first order in "`CONFIG_LSM`" of `.config` when compiling the kernel. SACK first executes the security check. After the check is allowed by SACK, they will be further checked by other security modules. To answer **Q3**, we test the compatibility with 10 different SACK policies for independent SACK and SACK-enhanced AppArmor, and they all work well with Ubuntu 20.04 default AppArmor policies. Because SACK utilizes the whitelist-based LSM stacking mechanism by setting `CONFIG_LSM`="SACK, AppArmor, ..." to ensure that SACK's security check performs before the other security modules. The other security modules perform access decisions only when SACK allows this access.

V. Conclusion and Acknowledgments

We propose SACK, a generic situation-aware access control mechanism in the Linux kernel. Compared with previous user space approaches, SACK is fine-grained, flexible, and hard to bypass. Our experimental results show that SACK can enforce situation-aware adaptive permissions with low runtime overhead. We also partly demonstrate SACK can achieve situation-aware permission adaptive enforcement and safety enhancement for CAVs in case studies. While we focus on CAVs in this paper, SACK is a general solution at kernel space and, therefore, applicable to scenarios such as the smartphone, IoT and medical application [38]–[40].

This work was supported by Huawei Technologies Co., Ltd. under Grant No. TC20210726015.

REFERENCES

[1] "Apparmor," https://www.apparmor.net, 2024.09.01.

[2] "Cve-2023-6073," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-6073, 2024.07.06.

[3] "Safe-driving," https://www.betterhealth.vic.gov.au/health/HealthyLiving/Safe-driving, 2023.10.02.

[4] "Visionfive2," https://www.starfivetech.com/en/site/boards, 2024.09.01.

[5] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, "Practical domain and type enforcement for unix," in *Proceedings 1995 IEEE Symposium on Security and Privacy*. IEEE, 1995, pp. 66–77.

[6] U. Bordoloi, S. Chakraborty, M. Jochim, P. Joshi, A. Raghuraman, and S. Ramesh, "Autonomy-driven emerging directions in software-defined vehicles," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[7] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 131–146.

[8] W. Cao, C. Xia, S. T. Peddinti, D. Lie, N. Taft, and L. M. Austin, "A large scale study of user behavior, expectations and engagement with android permissions," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 803–820.

[9] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "Crêpe: A system for enforcing fine-grained context-related policies on android," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 5, pp. 1426–1438, 2012.

[10] Corbet, "securityfs," https://lwn.net/Articles/153366/, 2024.09.10.

[11] G. Costantino and I. Matteucci, "Koffee-kia offensive exploit," *Istituto di Informatica e Telematica, Tech. Rep.*, 2020.

[12] W. Ding, H. Hu, and L. Cheng, "Iotsafe: Enforcing safety and security policy withreal iot physical interaction discovery," in *Network and Distributed System Security Symposium*, 2021.

[13] Z. Fang, W. Han, and Y. Li, "Permission based android security: Issues and countermeasures," *computers & security*, vol. 43, pp. 205–218, 2014.

[14] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses." in *USENIX security symposium*, vol. 30, 2011, p. 88.

[15] H. Fu, Z. Zheng, S. Zhu, and P. Mohapatra, "Keeping context in mind: Automating mobile app access control with user interface inspection," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 2089–2097.

[16] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," in *2016 46th annual IEEE/IFIP international conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 2016, pp. 215–217.

[17] S. Gansel, S. Schnitzer, A. Gilbeau-Hammoud, V. Friesen, F. Dürr, K. Rothermel, C. Maihöfer, and U. Krämer, "Context-aware access control in novel automotive hmi systems," in *Information Systems Security: 11th International Conference, ICISS 2015, Kolkata, India, December 16-20, 2015. Proceedings 11*. Springer, 2015, pp. 118–138.

[18] D. Grimm, M. Stang, and E. Sax, "Context-aware security for vehicles and fleets: A survey," *IEEE Access*, vol. 9, pp. 101 809–101 846, 2021.

[19] M. Gupta, F. M. Awaysheh, J. Benson, M. Alazab, F. Patwa, and R. Sandhu, "An attribute-based access control for cloud enabled industrial smart vehicles," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 6, pp. 4288–4297, 2020.

[20] D. K. Hong, J. Kloosterman, Y. Jin, Q. A. Cao, S. Mahlke, and Z. M. Mao, "Avguardian: Detecting and mitigating publish-subscribe overprivilege for autonomous vehicle systems," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 445–459.

[21] S. Jeong, M. Ryu, H. Kang, and H. K. Kim, "Infotainment system matters: Understanding the impact and implications of in-vehicle infotainment system hacking with automotive grade linux," in *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*, 2023, pp. 201–212.

[22] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, A. Prakash, and S. Unviersity, "Contexlot: Towards providing contextual integrity to appified iot platforms." in *ndss*, vol. 2, no. 2. San Diego, 2017, pp. 2–2.

[23] N. Malkin, A. F. Luo, J. Poveda, and M. L. Mazurek, "Optimistic access control for the smart home," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 3043–3060.

[24] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[25] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM symposium on information, computer and communications security*, 2010, pp. 328–332.

[26] S. Nie, L. Liu, and Y. Du, "Free-fall: Hacking tesla from wireless to can bus," *Briefing, Black Hat USA*, vol. 25, no. 1, p. 16, 2017.

[27] K. Olejnik, I. Dacosta, J. S. Machado, K. Huguenin, and J. P. Hubaux, "Smarper: Context-aware and automatic runtime-permissions for mobile devices," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[28] S. Owen, S. Wang, and P. Zhang, "Field assessment of gm/onstar occupant-based injury severity prediction," in *27th International Technical Conference on the Enhanced Safety of Vehicles (ESV) National Highway Traffic Safety Administration*, no. 23-0123, 2023.

[29] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi, "SEApp: Bringing mandatory access control to android apps," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3613–3630.

[30] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE communications magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[31] R. Schuster, V. Shmatikov, and E. Tromer, "Situational access control in the internet of things," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1056–1073.

[32] B. Shebaro, O. Oluwatimi, and E. Bertino, "Context-based access control systems for mobile devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 2, pp. 150–163, 2014.

[33] H. S. Varshith and Sural, "Efficiently supporting attribute-based access control in linux," *IEEE Transactions on Dependable and Secure Computing*, vol. 26, no. 6, pp. 1–15, 2023.

[34] H. S. Varshith, S. Sural, J. Vaidya, and V. Atluri, "Enabling attribute-based access control in linux kernel," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 1237–1239.

[35] D. Ward and P. Wooderson, *Automotive Cybersecurity: An Introduction to ISO/SAE 21434*. SAE International, 2021.

[36] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov, "The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 1077–1093.

[37] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the linux kernel," in *11th USENIX Security Symposium (USENIX Security 02)*, 2002.

[38] L. Xie, C. Li, Z. Wang, X. Zhang, B. Chen, Q. Shen, and Z. Wu, "Shisrcnet: Super-resolution and classification network for low-resolution breast cancer histopathology image," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2023, pp. 23–32.

[39] L. Xie, M. Lin, S. Liu, C. Xu, T. Luan, C. Li, Y. Fang, Q. Shen, and Z. Wu, "pflfe: Cross-silo personalized federated learning via feature enhancement on medical image segmentation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2024, pp. 599–610.

[40] L. Xie, M. Lin, T. Luan, C. Li, Y. Fang, Q. Shen, and Z. Wu, "Mh-pflid: Model heterogeneous personalized federated learning via injection and distillation for medical data analysis," in *Forty-first International Conference on Machine Learning*.

[41] D. Yu, G. Yang, G. Meng, X. Gong, X. Zhang, X. Xiang, X. Wang, Y. Jiang, K. Chen, W. Zou *et al.*, "Sepal: Towards a large-scale analysis of seandroid policy customization," in *Proceedings of the Web Conference 2021*, 2021, pp. 2733–2744.

[42] Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo, and E. Fernandes, "Moses: Supporting and enforcing security profiles on smartphones," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 3, pp. 211–223, 2014.

[43] W. Zhou, Y. Jia, Y. Yao, L. Zhu, P. Guan, and Y. Zhang, "Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 1133–1150.