Harness: Transparent and Lightweight Protection of Vehicle Control on Untrusted Android Automotive Operating System

Haochen Gong, Siyu Hong, Shenyi Yang, Rui Chang*,
Wenbo Shen, Ziqi Yuan, Chenyang Yu, and Yajin Zhou
Zhejiang University

Abstract

As modern in-vehicle infotainment (IVI) systems become more advanced and feature-rich, their complexity increases, expanding the attack surface. Since IVI systems often support vehicle controls, attackers can exploit their vulnerabilities to gain control of the car, posing a dangerous threat to property and personal safety. In this paper, we systematically analyze the attack surface of the Android Automotive Operating System (AAOS). We identify risks across the vehicle control chain, from the human-machine interface through relevant apps and services to the in-vehicle network communication. To prevent these risks, we propose Harness, a lightweight framework that transparently protects vehicle control from untrusted AAOS. Harness defines a minimal protection domain encompassing trusted software with permissions to perform security-critical vehicle control. Leveraging the hypervisor's capabilities, Harness isolates this domain from AAOS and protects its interactions with the external environment, ensuring vehicle control operations align with user intent. We implement Harness, and our evaluation shows it achieves security guarantees with only modest performance overhead.

1 Introduction

As the concept of software-defined vehicles (SDV) advances, in-vehicle systems are increasingly becoming connected and centralized, with multiple systems running on a single head unit via virtualization [60, 68]. Among these systems, the In-Vehicle Infotainment (IVI) system is widely deployed in modern vehicles. IVI systems are often connected to the vehicle's electronic control units (ECUs) through the in-vehicle network (IVN). This allows IVI to integrate vehicle control capabilities and offer users a more convenient and intelligent experience. One typical IVI system is the Android Automotive Operating System (AAOS) [34]. With Android's extensive ecosystem, AAOS provides abundant features and developer-friendly tools. However, Android's complexity introduces significant security issues. Android's trusted computing base (TCB) encompasses too many components, including the Linux kernel, device drivers, and userspace system services and apps. Besides, Android's allowance for third-party app installations further affects security. As shown in Figure 1, attackers could exploit vulnerabilities within these components, compromising the entire IVI and enabling unauthorized car control via the IVN. This poses a substantial risk to both property and personal safety. Therefore, **enhancing vehicle control security while retaining the rich software ecosystem and features of AAOS** is a significant research problem.

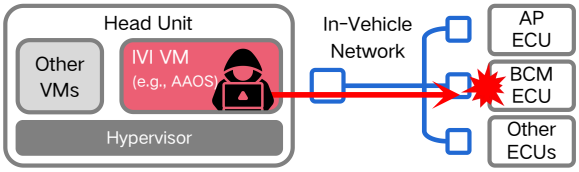


Figure 1: Attack from the IVI to other ECUs. AP means Autopilot, BCM means Body Control Module.

To reduce AAOS's TCB, a minimal protection domain can be constructed, including only the essential components (trusted) and isolating them from the rest of the system (untrusted). This way, the domain remains unaffected even if other system parts are compromised. Prior works have extensively studied the isolation of critical software from untrusted OSes. However, whether employing trusted execution environments (TEEs) [4, 9, 16, 20, 41, 83] or hypervisor-based protection [18, 33, 43, 46], they cannot be adapted to the scenario we face. First, most of them focus on isolating individual software components, such as an app or code piece, rather than protecting an entire control chain. Specifically, vehicle control in an IVI system typically involves human-machine interface (HMI), control decision, and IVN communication, which comprises a variety of kernel drivers and user-space programs. Due to their dependencies, attacks on any part of the chain could lead to losing control. Second, some approaches require hardware changes or non-trivial software

*The corresponding author.

modifications, which impose deployment burdens on vendors and even usage burdens on users. In particular, Android has lots of closed-source software [65, 74], such as libraries and drivers, which are hard to modify or decompose.

To this end, we present Harness, a lightweight framework for transparently protecting vehicle control on untrusted AAOS. Harness aims to ensure that vehicle controls initiated by the IVI align with user intentions. We define a minimal protection domain encompassing components with permissions to perform security-critical vehicle control. Based on the hypervisor, Harness leverages virtualization to create enclaves at process granularity to isolate the domain from the untrusted system. Harness establishes encrypted channels [67] between the domain and ECUs and prohibits software outside the domain from invoking security-critical car interfaces, thus preventing attackers from injecting malicious controls. By relying on the OS for complicated system features and inspecting critical operations, Harness can enclave unmodified programs while keeping a small TCB.

However, isolating unmodified software on the AAOS presents new challenges. First, *Android apps and services heavily rely on shared memory, conflicting with isolation.* Android supports flexible memory sharing across processes for better resource utilization (e.g., sharing thousands of libraries) and efficient cooperation [30]. Existing software-based works only support restricted memory sharing [18, 43, 44], like read-only and file-backed memory or sharing between parent-child processes or processes within the same app. None of these works support sharing writable anonymous memory between isolated domains, which is essential for Android framework.

Second, the most commonly used inter-process communication (IPC) mechanism in Android, *the Binder, is more complex and performance-sensitive than traditional Linux IPC*, e.g., pipe and socket. The Binder is frequently used during the lifecycle of an app or service to make API calls and transfer data. Hence, using cryptography employed by prior studies [16, 43, 70] to protect Binder would incur non-negligible overhead. Besides, Binder's data transmission is more complicated, making it unsuitable for end-to-end encryption. For instance, Binder supports transmitting data of complex types like Java objects, during which the kernel needs to adjust the pointers inside to the target address space.

Third, *Android's human-machine interface (HMI) is challenging to protect transparently*. As security-critical car interfaces are confined within the domain, adversaries can only perform related controls via the HMI, making its protection essential. Users mainly access the IVI via a touchscreen. Existing approaches [25, 55, 56] require additional hardware or extensive software changes, which burdens deployment and user experience. Furthermore, it is impractical to simply isolate all services the HMI depends on. Specifically, some of these services are also needed outside the domain, so it is impossible to completely isolate them. Moreover, services like the SystemServer have an enormous attack surface [58, 64, 75] and could be exploited to compromise other protected software that relies on them.

We propose three techniques to address the challenges above. First, the **sharing-aware memory monitor** leverages the hypervisor to monitor enclaves' intentions of memory mapping and introduces resource tokens to enforce access control, ensuring secure memory management while enabling controlled memory sharing. Second, the **Binder protection with lightweight authentication** tracks enclaves' registration and querying of Binder-based services and secures sensitive interface communications via the hypervisor. It also automatically generates call gates that collaborate with the hypervisor to authenticate communication parties. Third, the **transparent HMI validation** secures the HMI without relying on error-prone system services. It utilizes a mechanism named *virtio device binding* alongside the memory monitor to ensure the integrity of graphic user interfaces (GUIs) and validates the touch input is user initiation. Based on these technologies, Harness enables unmodified programs to run in enclaves and cooperate securely to perform vehicle controls.

We implement a Harness prototype based on the Google Cuttlefish virtual platform [39]. Our security analysis and evaluation show that Harness can defeat potential attacks from attackers that manipulate the IVI system. We further evaluate Harness's performance on a Raspberry Pi 5 board, demonstrating its efficiency and lightweight.

In summary, the contributions of this work are:

I. This is the first work to systematically study the attack surface of the AAOS's entire vehicle control chain. We identify and summarize 13 potential attack strategies originating from the kernel and userspace.

II. We propose a hypervisor-based framework that transparently and efficiently protects the vehicle control chain from the untrusted AAOS. The framework allows developers to enclave unmodified userspace components.

III. We implement Harness based on Google Cuttlefish and protect several prototype apps and services without modification, demonstrating compatibility and usability.

IV. We extensively analyze and evaluate Harness on real hardware platforms. Harness can defeat potential and real-world attacks while achieving efficiency and lightweight. It only incurs 3.35% and 6.09% slowdowns on Geekbench single- and multi-core tests and less than 4% memory overhead. Meanwhile, Harness does not affect the performance of unprotected software on the system.

2 Background

2.1 Virtualization in Automotive Systems

Modern vehicles usually adopt virtualization to reduce the number of electronic control units (ECUs) [60, 68]. Virtualization allows a single SoC to run multiple automotive systems such as In-Vehicle Infotainment (IVI) and Advanced

Driver Assistance Systems (ADAS). One implicit benefit of virtualization is to provide strong isolation between different systems and prevent them from interfering with each other.

Arm virtualization. Arm has a high market share in in-vehicle systems [5]. An Arm CPU core has four exception levels: EL0 for applications, EL1 for kernels, EL2 for hypervisors, and EL3 for secure monitor. The hypervisor at EL2 can configure the stage-2 page table of a Virtual Machine (VM) to control its view of memory. For address translation of a VM, the stage-1 paging translates the virtual address to the intermediate physical address (IPA), and subsequently, the stage-2 paging translates the IPA to the physical address. The IPA space is what the VM thinks is physical memory.

Virtio. Many automotive hypervisors [12, 29, 34] follow the virtio standard to realize I/O virtualization for performance-critical devices. Virtio provides an efficient para-virtualization framework that allows guests and the host to work cooperatively on I/O processing. The framework consists of *frontend (FE)* drivers in the guest and *backend (BE)* drivers in the host, where the latter can access physical devices through host drivers. Leveraging a *virtqueue*, an FE driver can communicate with the related BE driver. The virtqueue itself does not transfer I/O data directly but maintains descriptors that record attributes (e.g., IPA, size) of the guest's I/O buffers so that the host can access the memory of the guest directly.

2.2 Main Components of AAOS

Android architecture. Android is composed of multiple software layers. The bottom layer is a customized Linux kernel that manages the hardware and provides system-level services to userspace software. An Android app can be installed through an *Android Package (APK)* file. After launching, it runs one or more Linux processes. Inside an app's process, in addition to the app's logic, there also runs the *Android framework* and *Android runtime (ART)*, respectively responsible for providing Android API and executing bytecode. Userspace also runs numerous background processes of system services, such as the *SystemServer*. Apps can access system services through the Android framework's APIs based on the Binder IPC. Android provides the *ActivityManagerService (AMS)* in the SystemServer to manage components named `Activity`, which serve as screens for apps. For hardware management, Android uses a *Hardware Abstraction Layer (HAL)* to define standard interfaces for vendors to implement, allowing apps to ignore the underlying device drivers.

Binder IPC. The Binder is the most commonly used IPC mechanism in Android. Binder uses a client-server model, where the client process starts transactions to request services from the server process. Android introduces a kernel driver to manage these transactions. When a process creates a service, it constructs a Binder object correlated with a Binder node in the driver. To access a service, a process must first obtain its handle. Two processes can transfer a Binder object over an existing Binder connection, and the recipient will get the object's handle. Based on the method, there are two other ways to obtain handles. First, the *ServiceManager*, a service whose handle is 0 and available to all processes, provides Binder-based remote procedure calls (RPCs) to register and retrieve system services. Processes can call `addService` to register system services and `getService` or `checkService` to get handles. Second, the AMS is responsible for managing application services and also provides similar RPCs. In this case, processes can call `publishService` to register services and `bindService` to get handles.

During a transaction, the client calls the driver's `ioctl`, passing arguments that contain a handle. The driver then uses the handle to locate the target Binder node and its associated process. After finding the server process, the driver copies the client's data to a newly allocated buffer and maps it to a Binder-dedicated region within the server's address space, enabling the server to access it with only one data copy. During copying, the driver translates objects in the data from the client's context into the server's. For example, the driver will adjust pointers in the sender's address space to the target's or convert the sender's file descriptors (FDs) to the receiver's. Finally, a thread within the server process is woken up from the blocked `ioctl` to handle the request.

Android provides the *Android Interface Definition Language (AIDL)* to facilitate the interface definition of Binder-based service. To use AIDL, developers should define interfaces in `.aidl` files and implement them in the service code. On building, the AIDL compiler generates the fundamental code to use the Binder for both the client and server based on the `.aidl` file. This allows developers to focus solely on defining and implementing interfaces without caring about the underlying IPC mechanisms.

3 Vehicle Control Chain and Attack Surface

Threat model. We consider vulnerabilities within the IVI system's software, including third-party apps and the OS. A powerful attacker aims to take control of the car through the IVI system. The attacker can exploit vulnerabilities to compromise the OS. After gaining the root privilege, the attacker can bypass security features like SELinux and achieve arbitrary memory access and code execution. The attacker can also use confused deputy attacks like Iago attacks [17] from the malicious software to deceive car control components.

We assume all user software protected by Harness are bug-free and would not reveal sensitive data. Harness trusts the hypervisor as the code size of mainstream automotive hypervisors is comparable to Arm Trusted Firmware-A [7]. QNX [12] even adopts a microkernel design and has undergone several safety certifications. Therefore, we further assume the hypervisor has configured the IOMMU to defeat DMA attacks. We also assume the guest IVI OS can boot securely and is initially benign, allowing Harness to extract some metadata from it

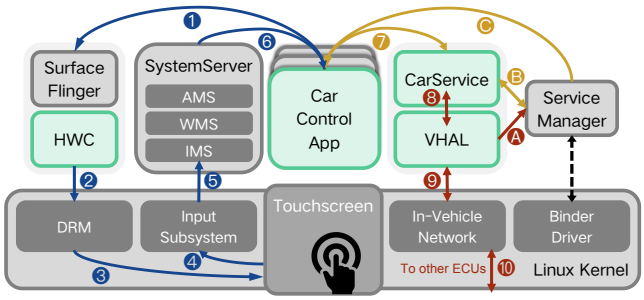


Figure 2: Car control chain of AAOS. Steps A-C indicate service registration and query rather than vehicle control.

and initialize properly. Physical attacks, hardware-based side channels, and denial-of-service attacks are out of the scope of this paper. We neither consider attacks against AI models (e.g., voice assistants and driver assistance) nor remote car control. All these problems are orthogonal and can be mitigated by combining Harness with other approaches.

Problem overview. Based on the threat model, we analyze the vehicle control chain in AAOS and its potential attack surface. Figure 2 outlines the chain, divided into three stages, i.e., the *HMI* stage (blue arrows), the *Process* stage (yellow arrows), and the *Output* stage (brown arrows). Table 1 summarizes 13 attack strategies, each can individually compromise the chain, resulting in losing control of vehicle operations. Some fundamental attack strategies, such as A1-A4, A7, and A9, apply to any component of the control chain. For example, attackers can leverage A1 to hijack the control flow of a process or A2 to break the memory integrity. The A3 can be launched to deceive a process to tamper with its stack and compromise the control flow. We also analyze attack strategies specific to AAOS and car control, including A5-6 that exploit inter-component dependencies, A8 and A10 that target critical kernel or system operations, and high-level strategies like HA1-3, built upon fundamental attack techniques.

3.1 The HMI Stage

The HMI Stage is the first stage, where the user locates a control widget (e.g., a button) on the touchscreen and makes a tap. These graphic user interface (GUI) widgets are provided by car control apps, which are pre-installed system apps rather than third-party apps. In Android, an app renders its GUI on a graphic buffer (namely `Surface`). Then, two services named *SurfaceFlinger* and *Hardware Composer (HWC)* collect all surfaces and compose them into the frame to be displayed (①). Finally, the HWC flushes the frame to the screen via the Linux *Direct Rendering Manager (DRM)* (②, ③). When the user touches the screen, the Linux input subsystem handles hardware interrupts and generates input events with data like coordinates (④). Afterward, the *InputManagerService (IMS)*, a service in the SystemServer, reads those events from the

Table 1: Attack Surface of vehicle Control in the IVI system.

Adversary	Attack Strategy	Affected Nodes/Steps	Requirements
Guest kernel	**A1**: Illegal register access	All processes	R2
	A2: Illegal page mapping	All processes	R2, R3, R4
	A3: Invalid syscall return (Iago)	All processes	R2, R3, R4
	A4: Device driver hijacking	All processes (I/O, IPC)	R3, R4
User process	**A5**: Service forging	①, ⑥, ⑦, ⑧, Ⓐ, Ⓑ, Ⓒ	R3, R4
	A6: Malicious data dispatching	①, ⑥, ⑦, ⑧, Ⓐ, Ⓑ, Ⓒ	R3, R4
Both	**A7**: Illegal memory access	All processes	R2, R3, R4
	A8: Unauthorized syscall	②, ⑨	R4, R1
	A9: Executable file tampering	All processes	R2
	A10: Config file tampering	CarService, VHAL	R2
	HA1: Unauthorized API call	⑦, ⑧	R3
	HA2: GUI confusion	①, ②, ③	R4
	HA3: Touch input forging	④, ⑤, ⑥	R4

The third column indicates the components/steps in Figure 2 that might be affected. R1-4 represent four basic requirements (in §3.4) to defeat these attacks.

kernel (⑤) and dispatches them to the target app depending on window states provided by the *WindowManagerService (WMS)* (⑥). Finally, the app receives input events and delivers them to the target UI components, namely `View`.

Potential attacks. Attackers can exploit the HMI stage to launch GUI confusion (HA2) [11, 31, 66, 85] or input injection (HA3) to manipulate vehicle controls. From the kernel (A4), attackers can directly tamper with the display and touch input data, including that in the memory of drivers (i.e., DRM and input, ③, ④), userspace software (i.e., system services and apps), and IPC (①, ⑥). Using A2, attackers can map an app's graphic buffer to another, leading to GUI confusion. More covertly, attackers may launch A3 to return fake touch events (⑤, ⑥) or graphic buffer handles to the userspace, leading to HA2. From the userspace, attackers who manipulate or impersonate system services (A5) can launch man-in-the-middle (MITM) attacks on HMI, including: (1) tampering with the HMI data passing through system services, either `Surface`s or input events (A7); (2) sending forged input events or `Surface`s to the car control app through the IMS/SurfaceFlinger (A6 on ⑥); (3) calling `ioctl` of DRM or input subsystem to manipulate rendering and display or inject input events (A8 on ②, ⑤); (4) dispatching input events on other foreground apps or GUI overlay to a background/underlying car control app (A6 on ⑥).

3.2 The Processing Stage

The processing stage is the second stage, where the car control apps and CarService process user inputs and make decisions on vehicle control. When the target `View` receives the inputs, it invokes car APIs exposed by the CarService (⑦). During creation, the CarService constructs several car-related services and registers itself to the ServiceManager (Ⓑ). When a car control app is launched, it instantiates an `Car` Java class, establishing a connection to the CarService through the ServiceManager (Ⓒ). CarService uses the AIDL to define interfaces, among which is the `ICar`. The `ICar` de-

fines an RPC named `getCarService` for apps to access the car-related services. For example, apps can use the RPC to get a handle referencing the *CarPropertyService*, the primary vehicle control service. The CarPropertyService also uses the AIDL to define its interface, which consists of RPCs that allow apps to get or set car properties like seat position and door locks. When such an RPC is called, it requests the Vehicle HAL (VHAL) to perform the hardware operations (⑧).

Potential attacks. Car control components within the processing stage increase the attack surface. To manipulate these components, attackers may tamper with APKs or binaries (A9), corrupt memory at runtime (A7, A2), or impersonate services (A5). These methods enable indirect injection of vehicle control commands via APIs exposed by CarService or VHAL (HA1 on ⑦, ⑧). Attackers can also exploit the ServiceManager to deliver malicious services (A5) to processes that make queries (A6 on ⑦). A malicious service can forge interfaces or send carefully constructed replies (e.g., invalid car property values) (Iago-like A6 on ⑦) to deceive the app into critical actions like displaying misleading information. Furthermore, attackers may tamper with configuration or preference files (A10) [63], causing the CarService and VHAL to load malicious car properties during initialization, potentially impacting the vehicle state.

Attackers can also exploit IPCs based on the Binder or shared memory. By compromising the Binder driver, they can tamper with transferred data (A4, A7) or map it to incorrect processes (A2). Additionally, attackers may invoke sensitive APIs (HA1) by injecting or replaying Binder requests (A8, A3). For shared memory, attackers can tamper with its content directly (A7) or manipulate page mappings (A2), such as mapping a shared page to unauthorized processes. This may cause the process to corrupt the page or unintentionally modify its content, compromising data integrity.

3.3 The Output Stage

The output stage is the final stage, where the VHAL interacts with ECUs through the in-vehicle network (IVN). The VHAL runs inside a standalone process and defines the car properties vendors can implement. It uses the AIDL to define interfaces and publishes itself in the ServiceManger (Ⓐ) so that the CarService can get its handle. The CarService can use the RPCs provided by the VHAL to get or set car property values. It is up to vendors to implement these RPCs, during which the VHAL should, in most cases, send car control requests to the vehicle bus through the IVN (⑩). ECUs connected to the bus will receive requests and perform specific controls.

Potential attacks. The output stage is the most vulnerable to attacks. The most straightforward attack is to inject car control commands into the IVN directly. To do so, attackers can manipulate the network stack or hardware controller (e.g., CAN controller) to send or replay packets containing such commands (A4). Or attackers can manipulate the process responsible for vehicle control (e.g., the VHAL) through system functions like `ptrace` to send commands [51] (A7 on ⑨ in Figure 2). Similarly, attackers can execute self-written code to invoke network-related syscalls (A8) to inject commands.

3.4 Security Boundary and Requirements

To defend against these potential attacks, we first need to identify the components that must be protected and mutually trusted, i.e., define a minimal protection domain. AAOS restricts IVN access exclusively to the VHAL and enforces access control on the car APIs using SELinux, allowing only permitted apps to invoke them [40]. Harness adopts and extends this security principle. Specifically, AAOS explicitly defines which car interfaces are security-critical and can only be accessed by trusted system components, i.e., the CarService and system apps for car control. Therefore, Harness confines these components and calls to these interfaces within the domain, imposing a clear security boundary with the external. The green components in Figure 2 represent the domain, comprising car control apps, the CarService, the VHAL, and the HWC, which are directly involved in vehicle control and HMI. Moreover, we duplicate a service named *Zygote* [30] into the domain, and Java programs within the domain should be forked from it. Components with extensive external interfaces, like the SystemServer, are excluded due to their susceptibility to attacks from third-party software [58, 64, 75]. To protect the domain, there are four essential requirements:

R1 Vehicle controls sent from the protection domain should be securely transferred to the target ECU. Software other than the domain should be forbidden from sending controls.

R2 The domain's memory and CPU registers should be loaded correctly and isolated from the untrusted OS. Pages within the domain should be mapped to the accurate address space and virtual address, preventing unexpected tampering.

R3 Processes within the domain can securely communicate with each other. Communications, including calls to security-critical interfaces, should not be forged, and the transmitted data should be protected.

R4 The HMI of vehicle control should be protected. Users should be able to see the correct car control GUIs, and their operations, such as tapping on the screen, should be correctly dispatched to car control apps.

4 Harness Overview

Harness aims to transparently secure the AAOS's vehicle control chain with an efficient and lightweight design. Figure 3 depicts the architecture of Harness. In a head unit, multiple VMs run on a hypervisor, with the IVI VM running AAOS being the protection target. We introduce a module named Harness Lowvisor within the hypervisor to inspect each vehicle control stage. Harness enclaves all the processes within the protection domain, carefully protecting their cooperation.

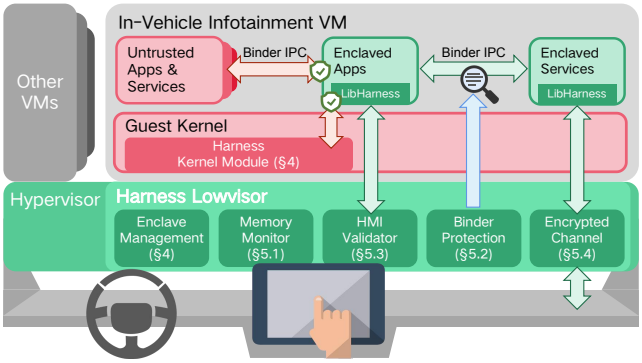


Figure 3: Harness architecture. Red components are untrusted, and green ones are trusted/protected.

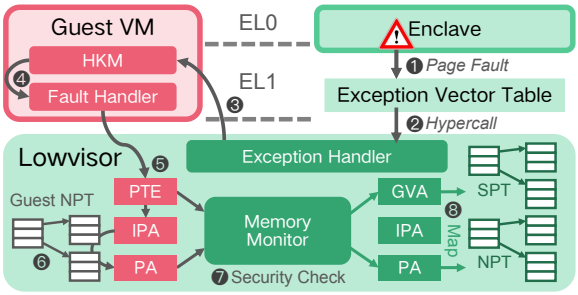


Figure 4: Page fault handling of Harness.

By establishing encrypted channels [67] between the domain and ECUs and enforcing access control on the domain interfaces, Harness satisfies **R1** (§5.4). The Lowvisor keeps a small code size by offloading complex tasks like scheduling and memory management to the guest OS. It interposes all transitions between enclaves and the OS, inspecting critical OS operations. While prior works can inspect common Linux semantics [43,44,83], Harness focuses on Android-specific semantics, including memory sharing, Binder IPC, and HMI on the touchscreen, which incurs new security and performance challenges. Therefore, we propose **three key techniques**, namely *sharing-aware memory monitor* (§5.1), *Binder protection with light authentication* (§5.2), and *transparent HMI validation* (§5.3), to satisfy **R2**, **R3**, and **R4**, respectively.

Harness enclave. Leveraging stage-2 paging, the Lowvisor creates enclaves at process granularity. Each enclave owns a separate user and kernel space, isolating data and code from the guest. The Lowvisor also maintains and protects an individual CPU context for each enclave, including EL0/EL1 system registers related to process control and general-purpose registers. Apart from the Lowvisor, we introduce a Harness kernel module (HKM) and a userspace library named LibHarness. The HKM bridges the gap between the guest kernel and enclaves. Concretely, it provides interfaces for the kernel to manage enclaves and dispatch exceptions from the enclaves to the kernel. The LibHarness encapsulates hypercalls provided by the Harness Lowvisor and `ioctl`s of HKM, making them available to developers. Developers from vendors do not need to change their code to use enclaves, as we modify the libraries in AOSP to use the LibHarness and make it transparent to the upper software layers.

Enclave lifecycle. A process can invoke the `eenter` function of LibHarness to enter an enclave. The library then calls the `ioctl` of the HKM, which manages the enclave via Lowvisor hypercalls. Initially, the HKM uses the `create_enclave` hypercall to instruct the Lowvisor to prepare the necessary enclave resources, including its CPU context and page tables, which are inaccessible to the guest OS. Subsequently, the HKM calls the `enter_enclave` hypercall, the Lowvisor then switches the CPU context and address space to those of the enclave and continues the process in the enclave. During execution, the Lowvisor intercepts the enclave's exceptions and forwards those trapped to EL1 to the kernel. To do this, the Lowvisor maps a customized exception vector table into the enclave's kernel space. This vector table, the only kernel-space component in the enclave, forwards exceptions to the Lowvisor via hypercalls. Harness configures the `VBAR_EL1` register in the enclave's CPU context to point to the vector table while the untrusted guest retains the original one.

The Lowvisor maintains two page tables for each enclave, i.e., a shadow page table (SPT) and a nested page table (NPT), as stage-1 and stage-2 page tables respectively. The two page tables are in the hypervisor's memory and inaccessible to the guest. Figure 4 shows how a page fault is handled. When it happens, the Lowvisor forwards it to the guest kernel (①-③). As normal, the kernel handles the fault and updates the original page table of the process (④). Then the HKM records the value of the updated page table entry (PTE) (⑤). Before the thread enters its enclave again, the Lowvisor walks the NPT of the guest to get the physical page and restricts the guest's access permission (⑥). After doing security checks (⑦, §5.1), the Lowvisor maps the page to the enclave's NPT and updates the SPT with the PTE value (⑧). The kernel needs to access an enclave's memory in two cases: the signal and syscalls that pass pointer arguments. Inspired by previous works [18,43,83], the Lowvisor creates each enclave thread with two shared buffers for the two scenarios, copying data between the buffers and the enclave memory as needed.

When destroying an enclave, Harness terminates all its threads and frees its resources in the Lowvisor and HKM. Pages no longer used by any enclave are returned to the guest, and anonymous ones are zeroed before that.

5 Design and Implementation

5.1 Sharing-aware Memory Monitor

Harness isolates processes' memory from each other and the untrusted OS and defeats A7. However, shared memory be-

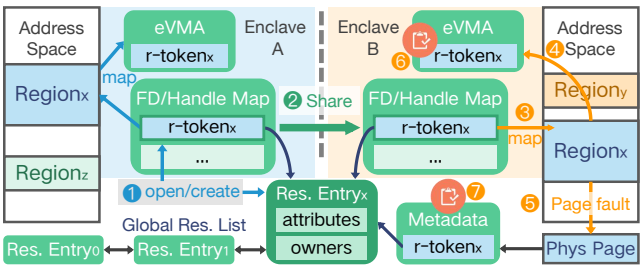


Figure 5: Main workflow of sharing-aware memory monitor. Res. represents for resource.

tween different processes, especially sharing writable anonymous memory, is an essential feature of Android [32, 37]. As mentioned in §3, allowing memory sharing while relying on the OS for memory management can cause severe security issues (A2-A3). Therefore, Harness should provide a secure memory-sharing mechanism for isolated processes while keeping compatibility and efficiency.

Shared memory in Android can be categorized into implicit and explicit sharing. Implicit sharing involves non-writable memory, such as library code or copy-on-write (CoW) memory. For this case, the Harness Lowvisor simply marks pages as write-protected by configuring NPTs of enclaves. Any process writing such a page triggers a CoW, where the Lowvisor assists in copying the data to a private page. Explicit sharing occurs when processes actively share memory through specific syscalls. For instance, a process can map the region as shared by setting a `MAP_SHARED` flag when calling `mmap`. Alternatively, the process can transmit the identifier of a memory resource to another process through IPC, usually the Binder in Android. Typically, there are two kinds of identifiers, the file descriptor (FD) and the handle. The former is commonly used for mechanisms such as `ashmem` and `memfd`, while the latter is widely used for I/O buffers like graphic buffers. When transmitting an FD, the kernel allocates a new FD referencing the same resource to the target process.

Security requirements. Harness must satisfy **R2** to securely share memory between enclaves. Specifically, it should ensure that memory mapping must be aligned with each enclave's intentions, e.g., a mapping's virtual address, length, and permissions, to defeat A2 and A3; enclaves can securely share identifiers of memory resources, and identifiers cannot be forged to defeat A2 and A3; and enclaves can map the correct page on page fault to defeat A2.

Intentions identification. Harness should first collect enclaves' intentions on memory mapping, including how memory can be shared. Inspired by Linux virtual memory area (VMA), the Lowvisor interposes on memory-related syscalls (i.e., `mmap`, `mprotect`, `mremap`, `munmap`, and `brk`), maintaining a set of simplified VMAs for each enclave, called eVMAs. When an enclave invokes one of these syscalls, the Lowvisor parses the arguments to obtain its intention on memory mapping and updates the corresponding eVMA. For example, when `mmap` is called, the Lowvisor can get from arguments the resource, offset, and virtual address range the enclave wants to map and the permission to set. On the syscall return, the Lowvisor checks whether the return value is consistent with the syscall's semantics and populates the related eVMA entry with the enclave's intention. With the eVMAs, the Lowvisor can determine whether the pages behind can be shared implicitly by checking the write permission, or shared explicitly by checking the `MAP_SHARED` flag.

To track the memory resources that can be shared through identifiers, the Lowvisor maintains a list (Figure 5's Global Res. List) to record information on all resources held by enclaves. Harness takes the address of each list entry as a unique token to identify the resource (named *r-token*). Enclaves holding the r-token have access to the resource. Each resource entry also has an owner list that records enclaves with access permission. Furthermore, the Lowvisor maintains an FD map and a handle map for each enclave, mapping FDs and handles to r-tokens, respectively. Figure 5 shows how this works. When an enclave calls `openat`, `memfd_create`, or ioctl that creates buffers, the Lowvisor parses the arguments and the return value to get the resource's attributes like its path, fills them to a newly allocated resource entry, and stores the r-token to the FD or handle map (①). Enclaves can securely share FDs or handles since Harness protects the Binder (§5.2). During this, the Lowvisor transfers the r-token from the sender enclave's FD/handle map to the receiver enclave's (②). When an enclave maps a resource via the FD, the r-token will also be stored at the corresponding eVMA entry (③,④).

Shareability validation. When a page fault occurs (⑤), the Lowvisor identifies which resource the enclave intends to access by searching for the eVMA entry the faulting address falls in and checking the r-token stored inside (⑥). To prevent malicious mapping, it further validates that the page allocated by the kernel belongs to the correct resource and is located at the proper offset. Specifically, the Lowvisor introduces additional *metadata* for each physical page. The metadata includes the token of the resource the page belongs to and the page's offset within the resource. When a page is first allocated to enclaves, the Lowvisor populates its metadata based on the eVMA and the faulting address. When the page is subsequently mapped again, the Lowvisor retrieves the metadata, compares the r-token inside with the one in the eVMA entry, and the offset with the one calculated from the eVMA and the faulting address (⑦). If all match, the Lowvisor considers it a legitimate page sharing and maps the page to the enclave. Harness only allows an anonymous page to be unmapped from an enclave when the enclave is destroyed or explicitly calls `munmap`.

Other benefits. The design of Harness also has other security and performance benefits. For example, with the eVMAs, the Lowvisor can validate the attribute bits of the PTEs allocated by the guest kernel (⑥ in Figure 5), preventing the kernel

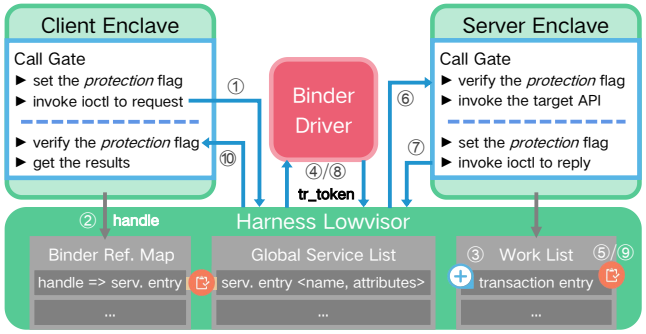


Figure 6: Binder protection and authentication. Ref. is short for reference. Serv. is short for service.

from tampering with the permission of pages (A2). Like previous works [18, 43], Harness also validates the integrity of file-backed pages through hash (§5.4) to defeat A9 and A10. The Lowvisor only needs to check for integrity the first time a page is mapped to enclaves. Then, the page can be securely shared between enclaves by checking the r-token and offset during page fault handling without recalculating the hash.

5.2 Binder Protection and Authentication

Binder is the essential mechanism for implementing car control APIs and therefore requires protection. Protecting Binder mainly faces two problems. First, the Binder is more complex and performance-sensitive than traditional Linux IPC. Hence, the cryptography-based approach commonly employed by existing works [16, 43, 70] is unsuitable for the Binder due to the significant adaptation effort and non-negligible performance overhead. Second, enclaves still need to communicate with components outside the protected domain. Overprotecting these communications may lead to excessive overhead and affect usability. Therefore, Harness needs a more efficient protection mechanism and a flexible way to identify which communications to be protected.

For flexibility, Harness allows developers to specify which Binder-based interfaces to protect. Specifically, we extend the AIDL to enable developers to mark interfaces as protected. Such interfaces can only be implemented by enclaved components and are protected by access control. Figure 6 shows how a protected Binder transaction works. We extend the transaction data structure with an additional field `prot_info` to store the metadata the protection needs. Our modified AIDL compiler generates a call gate for the caller of each protected interface. When an enclave invokes such an interface, its call gate notifies the Lowvisor by setting the `protection` flag. This flag, encapsulated in `prot_info`, is passed as part of the `ioctl` argument to the Binder driver (①).

Security requirements. To protect a marked Binder interface and meet **R3**, Harness must ensure that the interface's transaction should be directed to the intended recipient to defeat A2-3, A6 and HA1; the integrity of the transferred data should be guaranteed to defeat A7; and attackers cannot forge either side of a transaction to defeat A5-6, and HA1. To achieve the last one, Harness should verify that the caller and callee are mutually trusted, i.e., belong to the same protection domain. Hence, untrusted enclaves outside the domain and non-enclaved processes should be blocked.

Target determination. When the Lowvisor intercepts an `ioctl` starting a protected transaction, it must first identify the enclave providing the service to defeat A5. To achieve this, the Lowvisor maintains a global service list to record services published by enclaves. Each list entry stores a service's name and attributes, such as the identifier of its belonging enclave. When an enclave registers a service via RPC like `addService`, the Lowvisor extracts the information about the service and stores it in the service list. Similarly, when an enclave queries a service via RPC like `getService`, the Lowvisor interposes to parse the target service's name and handle. Then, the Lowvisor searches for the service's entry by name in the service list. If the entry is found, the Lowvisor stores its address in the enclave's binder reference map, which maps service handles to entries. These mechanisms prevent the untrusted ServiceManager or AMS from delivering inappropriate services (A5, A6). After an enclave invoking `ioctl` to start a transaction, the Lowvisor obtains the handle from the data passed to the driver. By retrieving the service entry from the binder reference map, the Lowvisor can resolve the target service and enclave (②). Then, the Lowvisor verifies whether the client and server enclaves belong to the same protection domain based on their identities (§5.4), blocking the transaction if they do not.

Transaction protection. To protect the transaction data efficiently, the Lowvisor transfers it between enclaves on behalf of the Binder driver, saving it from the sender and restoring it to the receiver. To do this, the Lowvisor maintains a work list for each enclave that has published a service. When a transaction is initiated, the Lowvisor allocates an entry in the work list of the target enclave (③). The entry (named transaction entry) is in the form of `<sender, metadata, state, async, data>`. The `metadata` specifies the target service and function being invoked, allowing the Lowvisor to validate against tampering during reception. The `state` field stores the current state of the transaction. Available states include `invalid`, `sent`, `waiting_for_reply`, and `replied`. The Lowvisor keeps close track of the state of each transaction to prevent the guest kernel from invoking protected interfaces through replay attacks (HA1 based on A3). The `async` field indicates whether the transaction is asynchronous (oneway). And the `data` field stores the data to transfer.

Harness takes the address of each entry as a token of the transaction (named *tr-token*). The tr-token is embedded into the `prot_info` within the `ioctl`'s argument and passed to the kernel. Before the recipient enclave receives the transaction from the kernel, the tr-token will be passed back to the

Lowvisor through the data returned by `ioctl`. If the tr-token belongs to the recipient enclave, the Lowvisor validates the transaction information based on the metadata stored in the work list of the enclave (⑤). Once validated, the Lowvisor restores the data from the work list to the recipient enclave and performs the object translation mentioned in §2.2 (⑥). When translating an FD, the Lowvisor checks whether the newly allocated FD is free. Following Binder's design, which provides each process with a memory region for data transfer, Harness introduces a shadow region for each enclaved process. The original region remains shared with the kernel and used by unprotected interfaces, while the protected ones use the shadow region. This enables Harness to protect transactions while introducing only one data copy and does not affect unprotected interfaces.

Lightweight authentication. Based on enclave identities (§5.4), Harness blocks protected interface calls initiated between mutually untrusted enclaves (②). Thus, if the Lowvisor retrieves a valid entry from an enclave's working list using the tr-token upon receiving a transaction, it confirms the transaction originated from a trusted enclave. Otherwise, the transaction is identified as coming from untrusted components. However, the Lowvisor's lack of interface-specific information makes it impossible to determine whether the transaction should be blocked at this stage. To address this, we propose an authentication mechanism that allows the Lowvisor and enclaves to collaborate in decision-making. Specifically, when building the server-side code of a protected interface, the modified AIDL compiler also generates a call gate. When a server enclave receives a transaction from a trusted enclave, the Lowvisor sets the `protection` flag in the `prot_info` field of the transaction data returned by `ioctl`. Otherwise, the flag is cleared. Once the control flow returns to the recipient enclave (⑥), its call gate validates the `protection` flag. If set, the call gate invokes the implemented interface function. Otherwise, the transaction is terminated. This protection and validation process also applies when the client enclave receives a reply from the server enclave (⑦-⑩). The difference is that the Lowvisor determines the target enclave by the `sender` field in the transaction entry rather than a handle.

5.3 Transparent HMI Validation

Since all security-critical car APIs are protected and confined within the domain, adversaries can only control the vehicle via the HMI, making its protection critical. Transparency is crucial for HMI protection in IVI systems. For users, protection mechanisms that complicate interactions (e.g., adding confirmation steps [77] or randomizing GUI layouts [56]) can distract the driver and compromise driving safety. For vendors, protection mechanisms that complicate product development and deployment (e.g., by introducing additional hardware [25] or requiring developers to refactor software [55]) will impede the efficiency of software iteration and OTA updates. Moreover, approaches rely on trusted remote servers [55, 78] fall short because vehicle control requires real-time feedback instead of intermittent connectivity and long latency.

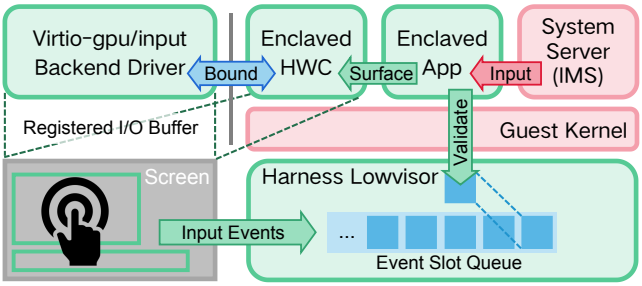


Figure 7: Transparent HMI validation.

With the support of the enclave and protected Binder, a straightforward design to achieve the transparency goal is enclaving all HMI-related components and establishing secure I/O channels with virtio BE drivers on the host. However, this can severely enlarge bloat the attack surface because components like SystemServer have a vast codebase and expose numerous interfaces, thus prone to vulnerabilities [58, 64, 75]. Therefore, we decide to exclude them from the protection domain. However, protecting HMI without trusting these services is challenging. As discussed in §3, attackers who control system services can easily launch an MITM attack on HMI.

Security requirements. To secure HMI and satisfy **R4**, Harness must ensure that car control apps' GUIs are displayed correctly to defeat A2-A8 and HA2; car control apps only react when the user touches screen areas within their GUIs to defeat A2-A8 and HA3; and the integrity of the user's touching event is not broken to defeat A2-A8 and HA3.

Virtio device binding. Harness introduces a mechanism called *virtio device binding* that allows an enclave to bind with a virtio device. Specifically, the mechanism enforces that a protected device can only work when binding with an HAL enclave. To do this, the Lowvisor provides two hypercalls named `bind_vdev` and `register_iobuff` for HAL enclaves to bind themselves with the corresponding virtual device and mark I/O buffers sharing data with the BE driver, respectively. Only registered buffers can be used for secure I/O, preventing malicious drivers from abusing the virtio framework.

Figure 7 shows how the mechanism protects HMI. For secure GUI display, Harness first enclaves the HWC and binds it to the `virtio-gpu`. We trust the HWC due to its limited interface exposure and compact codebase (about 6K LoC). When the HWC creates a frame buffer for display, it calls `register_iobuff`. The Lowvisor then unmaps the buffer's pages from the untrusted guest and provides the buffer's attributes to the BE driver, including the length and IPA of each page. Since apps render their GUIs, Harness also protects the buffers shared between enclaved car control apps and the `virtio-gpu`, including graphic buffers (`Surfaces`) and GPU command buffers. As the untrusted SurfaceFlinger man-

ages Surfaces, the Lowvisor further restricts that a protected Surface can only be mapped by an app and the HWC, preventing untrusted software from tampering with them. This is achieved by checking the owner list in the buffer's resource entry (§5.1). When apps finish rendering their GUIs of the current frame, the HWC composes their Surfaces into the frame buffer and flushes it to the virtio-gpu BE driver. The Lowvisor provides the HWC with the r-token when it imports a Surface of a car control app. Using the r-tokens, the HWC can identify the pixel areas associated with car control GUIs. Once an area changes, the HWC invokes the update_window hypercall to notify the Lowvisor of the updated position and boundary. With these attributes, the Lowvisor can determine whether a user's touch falls within a protected GUI.

Touch input validation. Since the IMS runs inside the error-prone SystemServer, we cannot enclave it and bind it with the virtio-input device. Instead, we still let the untrusted IMS process and dispatch all input events but validate the correctness of its behavior. Harness employs a match-based mechanism to verify that input events received by a car control app are user-initiated and fall within its GUI. The mechanism leverages the key insight that *the processing of input events by the IMS is reversible*. Specifically, as each raw event generated by the device only contains a single value like x-coordinate, the IMS first aggregates all events of a touch sampling, merging them into an event slot. Then, the input device coordinates are transformed into the display panel coordinates, where the computation is linear. Finally, the transformed event slot is constructed as an InputMessage populated with additional information. Therefore, we can infer the raw events based on the InputMessage received by the app.

In Harness, as shown in Figure 7, the Lowvisor collects raw touch events hitting the GUI areas of car control apps from the virtio-input BE driver. These events are constructed into raw event slots <x, y, action>, where action can be down, move, or up, and stored in per-app queues. When an InputMessage arrives at a car control app, the LibHarness inverts it back to the raw event slot and calls validate_input hypercall of the Lowvisor to validate it. The Lowvisor retrieves the corresponding original slots from the app's queue. If a matched slot is found, i.e., the x and y-coordinates and action are all equal, Lowvisor considers the event slot legitimate. Otherwise, Lowvisor notifies the app that an illegal event slot has been detected and should not perform any car control. Since input events are sequential, all event slots before the matched one are dropped to prevent event replay attacks. The Lowvisor also records the timestamp of the latest event in each queue to prevent malicious delays in dispatching events.

5.4 Trust Establishment

Since Android supports verified boot [35], we assume that the automotive system, including the IVI, can boot securely. After system boot, to meet **R1**, Harness needs to authenticate each enclave's identity to perform access control on them and establish encrypted channels connected to ECUs. For simplicity, like prior works [18, 43], all keys used by Harness are managed by the Lowvisor and protected from the OS.

Enclave authentication. Attackers may launch malicious enclaves. Therefore, Harness enforces authentication and access control to restrict their capability. To achieve this, each enclave's binaries are bound to a unique ID (named eID). We propose a two-step mechanism to validate this during enclave initialization. Note that Java-based components within the domain are system components. Hence, we AOT-compile them into the system ROM [36], treating them as binaries.

When deploying Harness, vendors must predefine a whitelist describing the protection domain by listing the protected components with their eID and metadata. This whitelist is only accessible to the Lowvisor. Before installing a component, Harness signs its binaries using the vendor's private key, during which the hash of the binaries (named bin_digest) is calculated. The signature and eID are recorded along with the binaries. Subsequently, Harness generates a secondary digest (named dig_digest) from the bin_digest and stores it in the corresponding entry of the whitelist. When starting an enclave, the OS provides its signature and eID to the Lowvisor. The Lowvisor retrieves the dig_digest from the whitelist using the eID to validate the bin_digest in the signature. Finally, the mapping from eID to bin_digest is established, and the enclave's integrity is validated when handling page faults. Since Java-based components are forked from a trusted enclaved Zygote (§3.4), Harness only needs to validate the newly loaded binaries, without repeatedly measuring shared resources such as the ART. The OS might provide a forged eID or signature. However, the two-step validation can detect anomalies and terminate in time, ensuring that enclaves can only load intended code and resources.

Based on the mechanism, Harness authenticates each enclave's identity and ensures that only the VHAL can connect to the IVN, preventing any other software from sending IVN messages. Combined with Binder-based authentication (§5.2), it also prohibits software outside the domain, whether enclaved or not, from accessing protected car interfaces.

Encrypted channel. Each ECU has the software stack to process IVN messages. Prior studies have proposed protecting IVN communication between ECUs via encryption [67], from which Harness can benefit. For simplicity, Harness uses symmetric encryption to establish the channel. During system boot, the Lowvisor provides a dedicated key to each ECU. When the enclaved VHAL starts, it obtains these keys from the Lowvisor. The VHAL signs a message before sending it to an ECU. The ECU then receives the message, decrypts and validates the signature, and activates the control. The ECU performs the same encryption process when replying to the VHAL. To defeat replay attacks, Harness generates a fresh nonce for each ECU. Each message records and increases the corresponding nonce (signed) during each transfer.

Table 2: The code size of Harness.

Module	Feature	LoC
Harness Lowvisor	Enclave management	1,897
	Syscall/signal support	1,873
	Memory management	950
	Resource monitor	858
	Binder protection	1,213
	HMI validation	248
	Others	484
	Total	7,523
HKM / Guest Kernel	Total / Modification	1,726 /173
LibHarness / AOSP	Total / Modification	79 / 914
Host Kernel	Modification	114

6 Evaluation

We implement a Harness prototype based on Google's open-sourced virtual platform named Cuttlefish [39], which uses KVM hypervisor [23, 24] to run AAOS in VMs. Cuttlefish takes crosvm [38] as its virtual machine monitor (VMM). Crosvm supports diverse virtual devices based on the virtio framework, including `virtio-gpu` and `virtio-input`. Note that the design principle of Harness is also applicable to other IVI systems. Table 2 shows the code size of our prototype. The main components of our prototype are about 9.3K lines of code (LoC). We add or modify about 914 LoC of AOSP to support our protection mechanisms. Besides, we only modify 287 LoC of the guest and host kernel to support Harness. In this section, we first compare our design with other solutions and explain why they are unsuitable for the target problem (§6.1). Then, based on the implementation, we extensively evaluate the security (§6.2), performance (§6.3), and usability (§6.4) of Harness.

6.1 Comparison with Existing Solutions

Table 3 illustrates a systematic comparison between Harness and related solutions. We exclude works that rely on unavailable hardware features on existing Arm platforms, like those based on hardware extensions [9, 21, 25, 27, 79, 80], Intel SGX [8, 10, 16, 20, 69, 70], or Arm CCA [3, 57, 72, 83]. Note that Harness is compatible with CCA, e.g., the Lowvisor and enclaves can run within the RMM and Realms, respectively.

Transparent protection of the AAOS vehicle control should meet the four security requirements (§3.4) while accommodating Android system semantics. Hence, four key features are desired, i.e., isolation that supports unmodified software, secure cross-enclave memory sharing, Binder-based inter-enclave communication, and secure HMI. Two main approaches to implementing the first feature are library OS-based [10, 16, 69] and exception-forwarding-based [18, 43, 83]. Harness adopts the latter, relying on untrusted OS services and performing security checks. This is because Android's inter-process dependencies make it impractical to isolate certain components completely. Packaging all dependent components into a single process would also undermine Android's built-in security mechanisms, such as SELinux. As detailed in §5, existing approaches fail to meet the last three goals due to functionality gaps, inefficiency, or deployment and usability challenges. Additionally, many Arm-based solutions [13, 19, 33, 42, 55, 56, 77, 78, 84] follow the TrustZone style that using machine-granularity isolation. This requires multiple protected processes to run on the same secure OS, e.g., OP-TEE [73] and Microdroid [6]. However, prior studies [14, 15] highlight the potential risk that all processes would be affected if the secure OS is compromised.

6.2 Security Analysis and Evaluation

TCB of Harness. The hypervisor and the Lowvisor residing in EL2 are the TCB of Harness. The design of Harness also applies to mainstream automotive hypervisors with a smaller code size than KVM. For example, ACRN [29] has a size of less than 40K LoC, comparable with Arm TF-A [7] (more than 30K LoC in v2.11). The code size of the QNX hypervisor [12] has not been disclosed, but it adopts a microkernel design and has undergone several safety certifications. Table 2 shows that the Harness Lowvisor only consists of 7,523 LoC, and therefore Harness does not significantly increase the TCB.

Defeat potential attacks. The last column of Table 1 summarizes how specific requirements can defeat those attacks. Overall, Harness isolates the protection domain from the OS (§4, 5.1) and ensures that only the domain can perform vehicle control (**R1**) by employing encrypted channels, enclave authentication (§5.4), and interface access control (§5.2). Externals can perform vehicle control only through the protection domain's HMI, which is also carefully protected (§5.3).

The Harness enclave and *sharing-aware memory monitor* (§5.1) meet **R2** and mitigate attacks A1-3, A7 and A9-10. Concretely, the Lowvisor maintains a CPU context for each enclave, handles the context switch, and thus prevents enclave registers from illegal access (A1). When handling `signals`, the Lowvisor validates the PC and ensures that enclaves can only jump to the registered handlers (A1). Harness isolates the physical pages of the protection domain to prevent A7 on it. The Lowvisor carefully inspects the guest kernel's memory operation, including page mapping and memory-related syscalls, to defeat A2 and memory-based A3 while allowing enclaves to share memory securely. It can also defeat controlled-channel attacks [76] since it prevents the guest kernel from directly managing enclaves' stage-1 page tables. Harness protects the integrity of critical files like executables and configurations, thus preventing A9 and A10.

The *Binder protection* (§5.2) meets **R3** and mitigates attacks A2-7 and HA1. Specifically, the Lowvisor tracks enclaves' registration and query of services to defeat attacks from the untrusted ServiceManger or AMS (A5-6). Based

Table 3: Systematic comparison of solutions adapt to existing Arm platforms. **Y** means the feature is supported, **P** means partially supported. "**OS semantics**" means the semantics provided to the isolated components. "**Unpriv. SW**" means the transparency to the unprivileged software development. Note that all these works require modifications on privileged software.

Category	Solutions	Mechanisms	Security Features				OS Semantics		Transparency	
			Domain Granu.	Mem Sharing	Inter-enclave Comm.	Secure HMI	Linux	Android	Unpriv. SW	User Expr.
Memory Model	Cerberus [52]	Monitor + Read-only	-	P	-	-	-	-	Y	-
HMI protection	Truz-Droid [77]	TEE (TrustZone)	-	-	-	Touchscreen	-	P	Y	-
	Truz-View [78]	TEE (TrustZone)	-	-	-	Touchscreen	-	P	Y	-
	TrustUI [56]	TEE (TrustZone)	-	-	-	Touchscreen	-	P	-	-
	VButton [55]	TEE (TrustZone)	-	-	-	Touchscreen	-	P	-	P
TEE / App protection	MyTEE [42]	Virtualization	VM	P	TA Session	Display & keyboard	-	-	Y	-
	AVF [33]	Virtualization	VM	Y	Binder & Linux IPC	-	Y	P	Y	-
	HyperEnclave [46]	Virtualization	Hybrid	-	-	-	-	-	Y	-
	Sanctuary [13]	TEE (TrustZone)	PM	P	Shared Memory	-	-	-	Y	-
	SecTEE [84]	TEE (TrustZone)	PM	P	TA Session	-	-	-	Y	-
	TrustShadow [41]	TEE (TrustZone)	Process	P	-	-	Y	-	Y	-
	OSP [19]	TEE (TrustZone) + Virt.	PM	-	-	-	-	-	Y	-
	Komodo [28]	TEE (TrustZone)	Process	-	-	-	-	-	Y	-
	Overshadow [18]	Virtualization	Application	P	Linux IPC	-	Y	-	-	-
	InkTag [44], Sego [49]	Virtualization	Process	P	Linux IPC	-	Y	-	-	-
	BlackBox [43]	Virtualization	Container	P	Linux IPC	-	Y	-	-	-
	Blindfold [54]	Monitor + PT switching	Process	P	-	-	Y	-	-	-
	Harness	**Virtualization**	**Process**	**Y**	**Binder & Linux IPC**	**Touchscreen**	**Y**	**Y**	**Y**	**Y**

Virt., PT, PM, Granu., Comm., Unpriv., Expr. and SW stand for Virtualization, Page Table, Physical Machine, Granularity, Communication, Unprivileged, Experience and Software.

Table 4: CVEs in IVI systems.

Type	CVEs
Privilege Escalation	2016-9337, 2018-9322, 2021-0303, 2021-39738, 2022-20144, 2022-42430, 2022-42431, 2023-20927, 2023-32155
Arbitrary Code Execution	2015-5611, 2019-9977, 2020-28656, 2021-23906, 2021-23907, 2021-23908, 2021-23909, 2023-32156, 2023-32157
Memory Corruption	2022-33300, 2022-40539
Tapjacking/Overlay	2021-0583, 2021-1036, 2021-1039, 2021-1040, 2022-20212, 2022-20214

on this, the Lowvisor further protects Binder transactions' data and dispatching, thus defeating A2-4, A6-7, and HA1. With the *lightweight authentication*, enclaves can identify each other and defeat A5 and HA1. Finally, the *transparent HMI validation* (§5.3) meets **R4** and mitigates attacks A2-8 and HA2-3. By binding the enclaved HWC with `virio-gpu`, a secure I/O channel is established to protect the integrity of the display (A2, A4-5, A7-8, HA2). By protecting the graphic buffers shared between the HWC and apps, Harness can identify the content on the display and further defeat GUI confusion (A2-7, HA2). Then, cooperating with the VMM and HWC, the Lowvisor can validate touch input and defeat HA1 which is based on A2-A8. Furthermore, due to the transparency, HMI protection does not affect personal safety.

Defeat real-world attacks. First, we analyze 26 CVEs related to IVI systems, including AAOS. Attackers can exploit vulnerabilities in Table 4 to escalate privilege, execute arbitrary code, corrupt OS memory, or leverage tapjacking/overlay attacks to confuse user interactions. Moreover, some of these CVEs have proven exploitable for vehicle control [1,2,50,51]. Since Harness strictly constrains vehicle control capabilities and protects the software within the protection domain, even if attackers exploit these vulnerabilities to compromise the IVI system, they cannot control the vehicle by injecting commands or manipulating protected software. We then analyze three attack paths proposed by a prior study [47], including attack from the IVI browser to the Body Control Module (BCM) ECU, expand code execution privileges from the IVI to the Autopilot ECU, and attack from IVI malware to car control. Similarly, Harness can defeat them by preventing malicious code from injecting car control commands.

Attack simulations. We further simulate several attacks targeting enclaved processes, including CPU context tampering (A1), malicious memory access (A7, HA2) and memory mapping (A2, A3), injecting touch input events (A4, A6, HA3), tampering with files (A9, A10), and invoking protected vehicle control interfaces (A8, HA1) or forging system services (A5). Consistent with our security analysis, Harness successfully detected all these malicious behaviors.

6.3 Performance Evaluation

We run experiments on a Raspberry Pi 5 with a 4-core Cortex-A76 64-bit 2.4 GHz Broadcom BCM2712 SoC, 8 GB RAM. The board runs our modified Raspberry Pi kernel (rpi-6.6.y) as the host kernel with KVM as the hypervisor. We use Google Cuttlefish to launch a guest IVI system with a configuration of 4 cores and 6GB RAM, running AOSP android-13.0.0_r41 with the common-android13-5.15 kernel. We first evaluate the performance overhead on individual system operations using `LMbench` and an app that measures Binder latency. We then assess the impact on user experience by measuring app startup

Table 5: LMbench results (μs). The lower the better.

Test case	Native	H-NE	Norm.	Harness	Norm.
null syscall	0.2474	0.2429	0.98	0.4975	2.01
read	0.3085	0.3011	0.98	0.6415	2.08
write	0.2855	0.2894	1.01	0.6186	2.17
stat	1.1949	1.2168	1.02	1.6401	1.37
fstat	0.6225	0.6191	0.99	1.1021	1.77
open/close	2.7691	2.8013	1.01	4.1279	1.49
select 100 fd	1.8072	1.8123	1.00	2.3837	1.32
signal install	0.297	0.2996	1.01	0.8098	2.73
signal catch	1.3486	1.3419	1.00	2.2697	1.68
pipe latency	16.559	17.369	1.05	22.070	1.33
AF_UNIX latency	14.7197	15.1809	1.03	17.964	1.22
fork+exit	219.5	233.8	1.07	28190	128.44
fork+execve	469.7	488.8	1.04	59201	126.05
fork+/bin/sh	6485	6556	1.01	105918	16.33
page fault	0.4575	0.4598	1.01	2.5653	5.61

Norm. shows the normalized performance. H-NE means Harness-NE.

Table 6: Startup time of an empty app (ms).

Type	Native	H-NE	Overhead	Harness	Overhead
Cold start	1234	1209	-2.0%	1410	13.5%
Hot start	310	328	5.8%	402	29.7%

Table 7: Latency of the input event (μs).

Input event	Native	H-NE	Over.	w/o IV	Over.	w/ IV	Over.
touch_down	1786.55	1762.20	-1.36%	1934.69	8.29%	1979.66	10.81%
touch_up	938.86	955.89	1.81%	1117.82	19.06%	1144.10	21.86%

Over. means overhead. IV means input validation.

Table 8: Memory consumption of Harness (MB).

System states	Native	Harness	Inc.	Overhead
Empty (Home)	4134.58	4252.96	118.37	2.86%
Car ctl. app (HVAC)	4139.94	4256.53	116.59	2.82%
Car ctl. app (API)	4137.86	4298.57	160.71	3.88%

Inc. means increment of memory usage.

time, input responsiveness, and car API latency. We also run a benchmark app to evaluate the overhead on typical workloads. Finally, we evaluate the Harness memory consumption to show its lightweight. To demonstrate Harness's impact on the rest of the system, we also evaluate the performance without protection (i.e., Harness-NE below, NE for non-enclave).

LMBench. We measure the performance of core OS operations using `LMBench 3.0` [61]. Table 5 shows performance measurements for each microbenchmark of Harness normalized to native execution. Most have less than or around two times the overhead. Note that fork and page fault incur relatively high overhead. The significant overhead introduced by the fork is due to creation of new enclaves. The 5.61x overhead imposed by page fault mainly results from the cost of TLB flush and additional maintenance on SPT and NPT.

Binder Latency. We develop an app and a service to measure the overhead of Binder protection. The app sends a message with varying payload sizes to the service through the Binder, and the service returns it intact. We repeat this round-trip 10,000 times and take the average latency. Figure 8a shows that the latency increases as the size of the payload increases. The overhead of Harness with or without Binder protection is about 60% compared to native execution. On average, the latter is just 4.6% slower than the former, thanks to only one additional data copy required by Harness. We also compare encryption-based protection methods used by prior works. Using AES to encrypt the data results in a 7.5x slowdown.

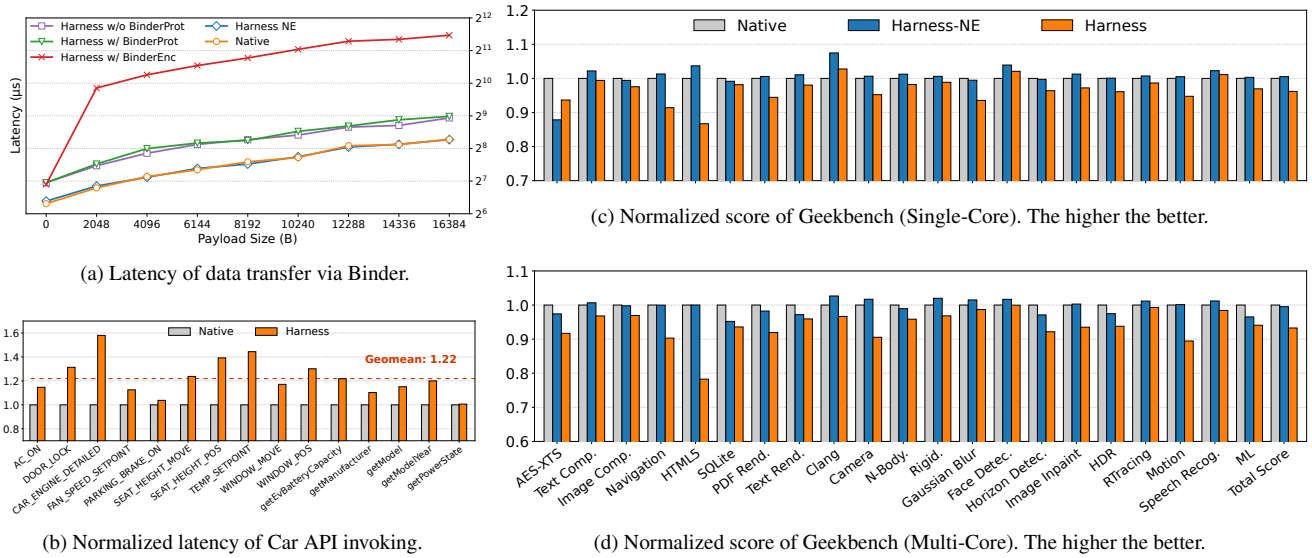
Startup time of apps. We measure the startup time of an empty app to demonstrate that, despite the high overhead of the fork operation, the responsiveness remains satisfactory for users. Table 6 shows that Harness results in a cold start slowdown of 13.53%, while the hot start is 29.68% slower. Even on a board with lower performance than the most advanced in-vehicle SoC, Harness only introduces a slowdown of less than 0.2 seconds, which is hardly noticeable to users.

Input responsiveness. To quantify the cost of input validation, we track the interval between the IMS InputDispatcher dispatching a touch event and the receiver app notifying the IMS for finish. A click on the screen causes IMS to send `touch_down` and `touch_up` events to the corresponding app. We calculate the average latency of these touch events after 1000 automatic clicks. Table 7 shows that the overhead is 10.81% (`touch_down`) and 21.86% (`touch_up`), among which the input validation only incurs 2.52% (`touch_down`) and 2.80% (`touch_up`) overhead. In addition, we test the touch event throughput, which could reach the device's upper limit of 60 events per second with input validation turned on.

Car API invoking. To measure the slowdown of car control commands, we select several Car APIs of the CarService involving cabin, sensor, air conditioning, etc., and develop an app to repeatedly invoke these APIs 5,000 times, record the latency, and take the average. Figure 8b presents the normalized results with a geometric mean overhead of 22%. Note that AOSP emulates these APIs and hence does not incur the latency of IVN communication, meaning the overhead would be much smaller in real-world deployments.

Benchmark Application. Figure 8c and 8d present the normalized single-core and multi-core performance results of `Geekbench 5.5.1` [45], respectively. The overhead of Harness on the total score is 3.35% for single-core and 6.09% for multi-core. The benchmark with the most significant slowdown is HTML5 (13.30% and 21.70% for single- and multi-core), primarily due to the frequent signal-related syscalls that lead to frequent context switching. Note that car control apps typically do not reach the high load of multi-core benchmarks. We do not evaluate the graphics performance since Cuttlefish does not support GPU acceleration on our board.

Impact on normal programs. Except for the API invoking



(a) Latency of data transfer via Binder.

(c) Normalized score of Geekbench (Single-Core). The higher the better.

(b) Normalized latency of Car API invoking.

(d) Normalized score of Geekbench (Multi-Core). The higher the better.

Figure 8: Performance overhead of Harness. (c) and (d) share the x-axis and legend.

test (since we protect these APIs from calling by untrusted apps), we also measure the overhead without protection on Harness. As shown in the figures and tables above, the performance of Harness-NE is nearly identical to that of the native. On average, the overheads of Harness-NE are 1.36%, 0.54%, -0.57%, and 0.46% for LMbench, Binder transmit, Geekbench single- and multi-core, which are negligible.

Memory consumption. Table 8 presents our analysis of memory consumption for Harness. We first evaluate the system's memory usage when no apps are running, during which services within the protection domain (e.g., the CarService) run in the background. Compared to the native system, Harness only introduces 2.86% additional memory consumption. For the case of running a car control app, Harness introduces a memory overhead of 2.82% to the system when launching the HVAC app and 3.88% when launching our API test app.

6.4 Usability of Harness

This section elaborates on the usability of Harness for vendors, users, and developers, followed by a discussion on Harness's applicability beyond vehicle control scenarios. **For vendors**, since Harness is software-based and involves minimal intrusive modifications to the system, it can be deployed to vehicles via OTA updates. The only hardware requirement is the widely supported virtualization. **For users**, as Harness protects unmodified car control apps, they will not experience any difference in usage. Moreover, the performance overhead of Harness is acceptable (§6.3), ensuring that the user experience remains seamless. **For developers**, adapting existing programs to Harness is straightforward. To enclave a Java program, developers only need to declare an XML element in the app's manifest, i.e., `AndroidManifest.xml`. To enclave a native program, developers only need to add a call to the LibHarness's `eenter` at the entry point. Alternatively, they can use our enclave loader, a program invokes `eenter` and then forks and executes the developer's program, similar to the Zygote. Overall, no more than 1 LoC needs to be added. To protect interfaces, developers only need to add an `hprot` mark before the interface definition in AIDL files, which can be done automatically using scripts. They can choose to modify the interface code to make finer use of the authentication information from the Lowvisor, i.e., the `protection` flag. Finally, a white list describing the trusted components should be provided to define a protection domain.

When protecting components other than vehicle control, Harness can safeguard components themselves, service registration and retrieval, cooperation between trusted components, critical interfaces, and the HMI. Harness may not cover all cases when enclaves consume data from untrusted components. It can benefit from prior studies [22, 71] to systematically analyze the potential security risks of untrusted Android API and implement security checks, which is our future work.

When protecting components that cannot be AOT-compiled and embedded in the system ROM, such as third-party apps, Harness can authenticate their enclaves by verifying the integrity of APKs using the method in §5.4. In this case, the `.oat` files (i.e., binaries) generated from `.dex` files via JIT compilation at runtime should also be protected. Alternatively, the system can be configured to perform AOT compilation during secure boot under specific conditions (e.g., after an OTA update) and secure the resulting binaries with signing-based integrity protection on the device.

Harness currently supports access control at the component

level but does not extend to finer-grained, such as specifying which component can access a specific interface. This design choice stems from the fact that security-critical interfaces are confined within the protection domain, where all components are mutually trusted. This reflects a trade-off between security functionality and Lowvisor complexity. If finer-grained access control is required in future problems, Harness can be easily extended. By defining rules in the whitelist to specify which enclaved components are allowed to access a protected interface, the Lowvisor can perform identity-based validation when the interface is invoked.

7 Related Work

Software protection on untrusted systems has been extensively studied by prior works. Sanctum [21], PENGLAI [27], CURE [9] and Keystone [53] are TEEs proposed for the RISC-V platform, the first three introduce new hardware features, while the latter leverages existing features to create enclaves. Some works use existing hardware TEE to support better protection. Graphene-SGX [16], SCONE [8], HAVEN [10], Occlum [69] and Panoply [70] support protecting unmodified apps in Intel SGX enclaves, either through library OSes or shim libraries. Sanctuary [13], Komodo [28] and SecTEE [84] leverage TrustZone to support SGX-like protection for Arm platforms. TrustShadow [41] proposes to run unmodified apps with Arm TrustZone. SHELTER [83] extends Arm CCA to enable isolation in userspace.

Other works [18,19,33,42–44,46,49,59,62,81,82] leverage virtualization to protect sensitive code and data from untrusted software with similar (or even higher) privileges, as well as from hardware attacks, which greatly inspired Harness.

Memory sharing between enclaves. Elasticlave [79] and Capstone [80] introduce hardware features to enable secure and efficient memory sharing. Software-based works only support restricted memory sharing between processes, such as read-only memory [52], file-backed memory [44], memory within an application [18], or memory shared between parent and child processes [43], which is insufficient to support Android's flexible sharing mechanisms.

IPC protection. Prior studies like Graphene-SGX [16], Panoply [70] and BlackBox [43] mainly focus on protecting traditional Linux IPC between enclaved processes. All these works leverage cryptography to prevent the untrusted kernel from accessing IPC data, which incurs non-negligible overhead. Instead, XPC [26] extends the hardware to support secure and efficient IPC, including Binder, without introducing additional data copy.

HMI protection. Several works use TrustZone's secure world to receive user input and display secure GUIs that overlay the original GUI [55,56,77,78]. Truz-Droid [77] and TruzView [78] require additional user confirmations to attest interactions. VButton [55] proposes a user-transparent attestation that captures snapshots on UI and then attests them via a remote server. TrustUI [56] randomizes the keyboard buttons and requires additional user clicks to defend against input tampering. Some studies propose new TEE (or TEE-like) mechanisms to protect HMI. MyTEE [42] proposes a TrustZone-style TEE that supports protecting the keyboard and display. PROTECTION [25] introduces an auxiliary device to display a secure UI overlay and receive user input.

8 Conclusion

Harness is a lightweight, transparent framework for protecting vehicle control on untrusted IVI systems. Harness can prevent attackers from gaining vehicle control from the IVI and ensure vehicle controls align with user intentions with modest overhead. The design and implementation of Harness seamlessly remain the user experience of IVI systems and can be easily adopted by vendors and developers.

Acknowledgments

We would like to thank our anonymous shepherd and reviewers for their insightful comments that greatly helped us to improve the paper. This work was supported by the National Key R&D Program of China (No. 2022YFB4501903), and the Key R&D Program of Zhejiang Province (No.2022C01165).

Ethics Considerations

The vast attack surface of in-vehicle infotainment systems may be exploited to compromise the vehicle, leading to its misbehavior. Our research focuses on strengthening the vehicle control chain to defeat potential attacks rather than finding new vulnerabilities. All the attack strategies we summarized in §3 are **from published studies**, and all the vulnerabilities we listed in §6.2 are **disclosed and fixed**.

For ethics considerations, we first identify the following stakeholders: malicious attackers, vehicle users, vehicle vendors that can be divided into the one we collaborate with, called $vendor_c$, and other vendors called $vendor_o$, as well as the potentially involved public, such as passengers and pedestrians. We then assess our research using consequentialist and deontological ethics as suggested by [48].

Consequentialist Ethics. Vehicle safety is a critical objective measure, as the compromise of the vehicle can pose significant threats to the physical and property safety of vehicle users and the public involved and jeopardize the interests of vendors, such as the need to withdraw faulty vehicles or compensate for damages.

Our study outlines the attack strategies and vulnerabilities within the IVI system. Attackers may attempt to refer to such information to perform attacks to the detriment of other stakeholders. However, as mentioned before, we did not discover any new vulnerabilities. We aim to provide a system-

atic summary of risks already known to the computer security community but may not be fully appreciated by all stakeholders, especially vendors. Vehicle users aware of these possible threats may have concerns and expect vendors to give security assurances. Vendors need to respond, and by fully understanding the identified risks, they can review what is lacking in their systems, consider and develop effective mitigations, and improve the overall security of their products.

In addition, our proposed protection framework, Harness, mitigates these identified attack strategies, making it much more difficult for attackers to compromise the vehicle. That minimizes potentially harmful impacts on vehicle users and provides a practical reference for $vendor_o$ to improve their vehicle security measures.

Deontological Ethics. The public and vendors have a right to be informed about potential security risks in the vehicle control chain. Transparency in this regard aligns with the ethical duty to respect the rights of individuals and organizations to make informed decisions. Since this work is conducted under a confidentiality agreement with $vendor_c$, obligating us to protect their commercial interests, we have agreed not to open-source certain critical components of the system to safeguard proprietary information.

Open science

We fully acknowledge the importance of open science policy in facilitating sharing and collaboration. In line with this, we are open to sharing the following artifacts[1]:

I. Source code of Harness prototype. Including the modified host and guest kernel, the Harness kernel module (guest), the Lowvisor (a host kernel module), the LibHarness, and a patch to the AOSP.

II. Source code for the attack simulation in §6.2. Including a userspace app and a patch to the Harness kernel module.

III. Source code for the performance evaluation in §6.3. Including the corresponding benchmark apps and scripts for analysis.

IV. Images of the prototype implementation. Since building AOSP can be time-consuming, we also provide prebuilt images for immediate use. These include image files of the modified Raspberry Pi host kernel, the modified guest android common kernel (containing the Harness kernel module), the host kernel module containing the Lowvisor, and the customized AOSP image configured for automotive (i.e., AAOS). We also provide images of the baseline.

With the published artifacts, all experiments are reproducible, including the performance evaluation in §6.3 and the attack simulation in §6.2. Due to inevitable variations in the experimental environment, minor acceptable deviations may occur.

[1] https://doi.org/10.5281/zenodo.14723474

References

[1] America's Cyber Defense Agency. Tesla gateway ecu vulnerability. https://www.cisa.gov/news-events/ics-advisories/icsa-16-341-01, 2016.

[2] America's Cyber Defense Agency. Harman-kardon uconnect vulnerability. https://www.cisa.gov/news-events/ics-advisories/icsa-15-260-01, 2018.

[3] Arm. Arm confidential compute architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2021.

[4] Arm. Arm trustzone technology. https://developer.arm.com/ip-products/security-ip/trustzone, 2021.

[5] Arm. Co-creating the future of vehicle electronics: Challenges and opportunities. https://www.arm.com/campaigns/vehicle-electronics-ford, 2021.

[6] Arm. Microdroid. https://source.android.com/docs/core/virtualization/microdroid, 2024.

[7] Arm. Trusted firmware-a. https://www.trustedfirmware.org/projects/tf-a/, 2024.

[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, D. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[9] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with CUstomizable and resilient enclaves. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1073–1090. USENIX Association, August 2021.

[10] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33:1 – 26, 2014.

[11] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. What the app is that? deception and countermeasures in the android user interface. In *2015 IEEE Symposium on Security and Privacy*, pages 931–948. IEEE, 2015.

[12] BlackBerry. QNX hypervisor virtualization software. https://blackberry.qnx.com/en/products/foundation-software/qnx-hypervisor, 2024.

[13] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'19)*, 2019.

[14] David Cerdeira, José Martins, Nuno Santos, and Sandro Pinto. ReZone: Disarming TrustZone with TEE privilege reduction. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2261–2279, 2022.

[15] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[16] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[17] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.

[18] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review*, 42(2):2–13, 2008.

[19] Yeongpil Cho, Junbum Shin, Donghyun Kwon, MyungJoo Ham, Yuna Kim, and Yunheung Paek. Hardware-Assisted On-Demand hypervisor activation for efficient security critical code execution on mobile devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 565–578, Denver, CO, June 2016. USENIX Association.

[20] Intel Corporation. Intel software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014.

[21] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[22] Rongzhen Cui, Lianying Zhao, and David Lie. Emilia: Catching iago in legacy code. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'21)*, 2021.

[23] Christoffer Dall, Shih-Wei Li, and Jason Nieh. Optimizing the design and implementation of the linux ARM hypervisor. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 221–233, Santa Clara, CA, July 2017. USENIX Association.

[24] Christoffer Dall and Jason Nieh. Kvm/arm: the design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*, page 333–348, New York, NY, USA, 2014. Association for Computing Machinery.

[25] Aritra Dhar, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capkun. Protection: Root-of-trust for io in compromised platforms. In *Proceedings 2020 Network and Distributed System Security Symposium (NDSS 2020)*, volume 2, pages 1377–1394. Internet Society, 2020.

[26] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. Xpc: architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 671–684, 2019.

[27] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 275–294. USENIX Association, July 2021.

[28] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 287–305, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Linux Foundation. ACRN: A big little hypervisor for iot development. https://projectacrn.org, 2024.

[30] Google. Overview of memory management. https://developer.android.com/topic/performance/memory-overview, 2023.

[31] Google. Tapjacking. https://developer.android.com/privacy-and-security/risks/tapjacking, 2023.

[32] Google. Android graphics. https://source.android.com/docs/core/graphics, 2024.

[33] Google. Android virtualization framework (avf). https://source.android.com/docs/core/virtualization, 2024.

[34] Google. Aosp automotive. https://source.android.com/docs/automotive/virtualization, 2024.

[35] Google. Aosp verified boot. https://source.android.com/docs/security/features/verifiedboot, 2024.

[36] Google. Configure art. https://source.android.com/docs/core/runtime/configure#system_rom, 2024.

[37] Google. Content providers. https://developer.android.com/guide/topics/providers/content-providers, 2024.

[38] Google. crosvm - the chromeos virtual machine monitor. https://chromium.googlesource.com/chromiumos/platform/crosvm/, 2024.

[39] Google. Cuttlefish virtual android devices. https://source.android.com/docs/devices/cuttlefish, 2024.

[40] Google. Vehicle system isolation. https://source.android.com/docs/automotive/security/vehicle_system_isolation, 2024.

[41] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 488–501, 2017.

[42] Seung-Kyun Han and Jinsoo Jang. Mytee: Own the trusted execution environment on embedded devices. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'23)*, 2023.

[43] Alexander Van't Hof and Jason Nieh. BlackBox: A container security monitor for protecting containers on untrusted operating systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 683–700, Carlsbad, CA, July 2022. USENIX Association.

[44] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 265–278, 2013.

[45] Primate Labs Inc. Geekbench 5 cpu workloads. https://www.geekbench.com/doc/geekbench5-cpu-workloads.pdf, 2019.

[46] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. HyperEnclave: An open and cross-platform trusted execution environment. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 437–454, Carlsbad, CA, July 2022. USENIX Association.

[47] Pengfei Jing, Zhiqiang Cai, Yingjie Cao, Le Yu, Yuefeng Du, Wenkai Zhang, Chenxiong Qian, Xiapu Luo, Sen Nie, and Shi Wu. Revisiting automotive attack surfaces: a practitioners' perspective. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 80–80. IEEE Computer Society, 2023.

[48] Tadayoshi Kohno, Yasemin Acar, and Wulf Loh. Ethical frameworks and computer security trolley problems: Foundations for conversations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5145–5162, Anaheim, CA, August 2023. USENIX Association.

[49] Youngjin Kwon, Alan M. Dunn, Michael Z. Lee, Owen S. Hofmann, Yuanzhong Xu, and Emmett Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 277–290, New York, NY, USA, 2016. Association for Computing Machinery.

[50] Tencent Keen Security Lab. Experimental security assessment of bmw cars. https://keenlab.tencent.com/en/whitepapers/Experimental_Security_Assessment_of_BMW_Cars_by_KeenLab.pdf, 2019.

[51] Tencent Keen Security Lab. Experimental security assessment of mercedes-benz cars. https://keenlab.tencent.com/en/2021/05/12/Tencent-Security-Keen-Lab-Experimental-Security-Assessment-on-Mercedes-Benz-Cars/, 2021.

[52] Dayeol Lee, Kevin Cheang, Alexander Thomas, Catherine Lu, Pranav Gaddamadugu, Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A Seshia, and Krste Asanovic. Cerberus: A formal approach to secure and efficient enclave memory sharing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1871–1885, 2022.

[53] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on*

Computer Systems, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[54] Caihua Li, Seung-seob Lee, and Lin Zhong. Blindfold: Confidential memory management by untrusted operating system. *arXiv preprint arXiv:2412.01059*, 2024.

[55] Wenhao Li, Shiyu Luo, Zhichuang Sun, Yubin Xia, Long Lu, Haibo Chen, Binyu Zang, and Haibing Guan. Vbutton: Practical attestation of user-driven operations in mobile apps. In *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, pages 28–40, 2018.

[56] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, pages 1–7, 2014.

[57] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and verification of the arm confidential compute architecture. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 465–484, 2022.

[58] Baozheng Liu, Chao Zhang, Guang Gong, Yishun Zeng, Haifeng Ruan, and Jianwei Zhuge. FANS: Fuzzing android native system services via automated interface analysis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 307–323. USENIX Association, August 2020.

[59] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619, 2015.

[60] Marketsandmarkets. Automotive hypervisor market by type, vehicle type, end user, level of autonomous driving, bus system (controller area network (can), local interconnect network (lin), ethernet, and flexray), sales channel and region - global forecast to 2027. https://www.marketsandmarkets.com/Market-Reports/automotive-hypervisor-market-124958216.html, 2023.

[61] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, page 23, USA, 1996. USENIX Association.

[62] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*, pages 157–171, 2020.

[63] Mert Pese, Kang Shin, Josiah Bruner, and Amy Chu. Security analysis of android automotive. *SAE International Journal of Advances and Current Practices in Mobility*, 2(2020-01-1295):2337–2346, 2020.

[64] Andrea Possemato, Dario Nisi, and Yanick Fratantonio. Preventing and detecting state inference attacks on android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'21)*, 2021.

[65] Qualcomm. Qualcomm linux graphics guide. https://docs.qualcomm.com/bundle/publicresource/topics/80-70014-19/graphics-overview.html, 2024.

[66] Chuangang Ren, Peng Liu, and Sencun Zhu. Windowguard: Systematic protection of gui security in android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[67] Kui Ren, Qian Wang, Cong Wang, Zhan Qin, and Xiaodong Lin. The security of autonomous driving: Threats, defenses, and future directions. *Proceedings of the IEEE*, 108(2):357–372, 2019.

[68] Grand View Research. Automotive hypervisor market size, share trends analysis report by product, by vehicle, by end use (economy, mid-priced, luxury), by mode of operation, by region, and segment forecasts, 2024 - 2030. https://www.grandviewresearch.com/industry-analysis/automotive-hypervisor-market#, 2024.

[69] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 955–970, New York, NY, USA, 2020. Association for Computing Machinery.

[70] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[71] Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. BesFS: A POSIX filesystem for enclaves with a mechanized safety proof. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 523–540, 2020.

[72] Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde. ACAI: Protecting accelerator execution with arm confidential computing architecture. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3423–3440, 2024.

[73] TrustedFirmware. Op-tee. https://www.trustedfirmware.org/projects/op-tee/, 2024.

[74] Wikipedia. Android (operating system). https://en.wikipedia.org/wiki/Android_(operating_system)#cite_note-6, 2024.

[75] Xiaobo Xiang, Ren Zhang, Hanxiang Wen, Xiaorui Gong, and Baoxu Liu. Ghost in the binder: Binder transaction redirection attacks in android system services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1581–1597, 2021.

[76] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[77] Kailiang Ying, Amit Ahlawat, Bilal Alsharifi, Yuexin Jiang, Priyank Thavai, and Wenliang Du. Truz-droid: Integrating trustzone with mobile operating system. In *Proceedings of the 16th annual international conference on mobile systems, applications, and services*, pages 14–27, 2018.

[78] Kailiang Ying, Priyank Thavai, and Wenliang Du. Truz-view: Developing trustzone user interface for mobile os using delegation integration model. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pages 1–12, 2019.

[79] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. Elasticlave: An efficient memory model for enclaves. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4111–4128, 2022.

[80] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E Carlson, and Prateek Saxena. Capstone: a capability-based foundation for trustless secure memory access. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 787–804, 2023.

[81] Ziqi Yuan, Siyu Hong, Rui Chang, Yajin Zhou, Wenbo Shen, and Kui Ren. Vdom: Fast and unlimited virtual domains on multiple architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 905–919, 2023.

[82] Ziqi Yuan, Siyu Hong, Ruorong Guo, Rui Chang, Mingyu Gao, Wenbo Shen, and Yajin Zhou. Lightzone: Lightweight hardware-assisted in-process isolation for arm64. In *Proceedings of the 25th International Middleware Conference*, pages 467–480, 2024.

[83] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. SHELTER: Extending arm CCA with isolation in user space. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6257–6274, Anaheim, CA, August 2023. USENIX Association.

[84] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. Sectee: A software-based approach to secure enclave architecture using tee. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1723–1740, New York, NY, USA, 2019. Association for Computing Machinery.

[85] Hao Zhou, Shuohan Wu, Chenxiong Qian, Xiapu Luo, Haipeng Cai, and Chao Zhang. Beyond the surface: Uncovering the unprotected components of android against overlay attack. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'24)*, 01 2024.