Head Unit at Risk: Cross-Domain Attacks via In-Vehicle Infotainment Systems

Yanbo Xu
Shanghai Jiao Tong University
Shanghai, China
yanbo_xu@sjtu.edu.cn

Yan Meng
Shanghai Jiao Tong University
Shanghai, China
yan_meng@sjtu.edu.cn

Guoxing Chen
Shanghai Jiao Tong University
Shanghai, China
guoxingchen@sjtu.edu.cn

Haojin Zhu*
Shanghai Jiao Tong University
Shanghai, China
zhu-hj@sjtu.edu.cn

Abstract

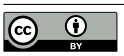
The rising intelligence of in-vehicle infotainment (IVI) systems enables them to provide advanced services like smartphones. Instead of manual operation, the IVI system features a touch-enabled *head unit* with various apps, allowing users to perform tasks easily (e.g., click in-app open trunk button). While expanded capabilities introduce security risks, manufacturers aim to mitigate them through strict permission controls and partitioning in-vehicle components into isolated domains to prevent attacks against each other.

This study, however, challenges the assumption that these mechanisms provide reliable security guarantees. Motivated by the observation that privileged apps possess cross-domain access capabilities and that operating systems (OSs) are typically derived from Android, we exploit native Android analysis surfaces to uncover potential vulnerabilities. Initially, we develop IVIHunter, an automated analysis framework designed to reverse-engineer proprietary in-vehicle network protocols without physical vehicle access. IVIHunter triggers vehicle control actions based on program analysis and correlates them with underlying traffic signals sent from the head unit, effectively mapping high-level vehicle control logic to low-level commands. Experiments on 4 head units show IVIHunter identifies 230 vehicle control functions across different domains and 629 corresponding CAN messages. Driven by IVIHunter, we launch two cross-domain attacks targeting a BYD vehicle: the remote vehicle status monitoring and malicious vehicle control command injection. The former achieves remote monitoring of 26 types of vehicle status, potentially compromising drivers' interests, while the latter realizes the manipulation of 10 driver assistance features, which significantly threatens driving safety.

CCS Concepts

• **Security and privacy → Security in hardware**.

*Haojin Zhu is the corresponding author.



CCS '26, The Hague, Netherlands

ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832669

Keywords

Smart vehicle, In-vehicle infotainment system, Cross-domain attack

ACM Reference Format:
Yanbo Xu, Yan Meng, Guoxing Chen, and Haojin Zhu. 2026. Head Unit at Risk: Cross-Domain Attacks via In-Vehicle Infotainment Systems. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832669

1 Introduction

The in-vehicle infotainment (IVI) system has emerged as a primary interface through which drivers interact with smart vehicles. The market size of IVI systems reached $35.78 billion in 2025 and is projected to $66.12 billion by 2034 [48]. A key component of the IVI system, the *head unit*, resembles a tablet (Figure 1) and supports smartphone-like user interaction. In its early stages, the head unit supported only basic functions (e.g., listening to the radio, playing CDs). Now, it has evolved into a mobile platform where various in-vehicle apps can be installed just like apps on Android. These apps have access to a wide range of physical in-vehicle components, including the vehicle body, chassis, and engine [22, 42, 47, 84]. For example, the head units in BYD (one of the world's largest electric vehicle manufacturers) and Lynk&Co (co-developed by Geely and Volvo) vehicles include a car settings app that enables drivers to monitor vehicle status (e.g., speed, energy consumption) and adjust configurations such as assisted driving and driving modes.

However, the head unit's increasing capabilities have been exploited for severe attacks [26, 28, 54, 61, 62, 66, 77]. For example, Miller and Valasek [62] achieved malicious vehicle control via the head unit in Black Hat 2015. To mitigate various attacks, SAE J3061 [1], which is also integrated into ISO/SAE 21434 [49], specifies that the in-vehicle network should adopt a domain controller-based hardware architecture, namely domain-centralized electrical/electronic (E/E) architecture [11, 12, 16, 82], to replace the distributed architecture targeted by Miller and Valasek's attack. In this architecture, in-vehicle components (e.g., engine, brake) are partitioned into independent domains based on functionality, such as infotainment, chassis, powertrain, body, and advanced driver-assistance system (ADAS), and the case of BYD Qin is illustrated in Figure 1. Additionally, a robust access control is implemented at the central gateway between domains and at controllers within each domain [15, 53, 65],

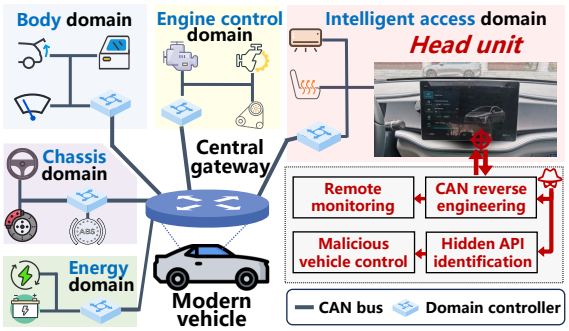


Figure 1: Domain-centralized architecture within the BYD Qin Plus and attacks driven by IVIHunter.

strictly restricting cross-domain control and thereby effectively preventing attacks launched by compromised components against other domains. For example, an illegal braking command sent from the body domain to the chassis domain would be blocked by the central gateway. Moreover, cross-domain access is also managed in in-vehicle components. For instance, only privileged system apps in head unit are granted such capability. The domain-centralized architecture has been adopted by numerous key manufacturers, such as Volkswagen, Toyota, General Motors (GM), Hyundai, BYD, Ford, Honda, BMW, Volvo, and Lynk&Co [23, 38, 74], and McKinsey estimates that it will be adopted by 48% of vehicles by 2030 [18].

Research motivations. In this paper, we aim to challenge the assumption that the aforementioned mechanisms ensure the security of modern vehicles. Our study is motivated by two key observations. (1) To enable vehicle control and status retrieval functions, privileged apps on the head unit are granted permissions to access and control components in other domains. Such cross-domain capabilities suggest that the head unit could serve as a potential attack surface. (2) The operating systems (OSs) running on head units are often adapted from well-established systems. For example, leading manufacturers such as Volkswagen, General Motors, Hyundai, Ford, Nissan, Honda, Volvo, and Lynk&Co employ Android Automotive OS (AAOS) [43] based customized OSs [3, 17, 27, 29, 37, 46, 80, 85], while automakers like BYD, BMW, and Geely also develop OSs built on Android [14, 22, 94]. This adaptation creates a potential opportunity, as analysis surfaces targeting AAOS or Android remain exploitable in vehicles. However, to launch attacks from the head unit, two research questions (RQs) must be addressed:

- **RQ1: How can we uncover the working principles of the head unit and reveal how they may expose potential vulnerabilities that enable cross-domain attacks?** It is crucial to understand the underlying mechanisms by which it accomplishes cross-domain vehicle control. In addition, it remains unclear whether the closed-source nature of the target OSs hinders automated analysis and launching severe attacks.
- **RQ2: How can we demystify proprietary in-vehicle protocols solely through analysis of the head unit, and subsequently leverage them to launch attacks such as monitoring and injection?** Most manufacturers use proprietary in-vehicle network protocols, which are commonly built on the controller area network (CAN) [2]. Without parsing protocol details, particularly uncovering the semantic details of CAN messages (i.e., through reverse engineering), meaningful monitoring or message injection becomes infeasible. However, existing approaches face significant challenges in our scenario: they are rendered impractical due to extensive manual operations [8, 32, 60, 71], rely on costly specialized equipment and real vehicles [90], and struggle with semantic recovery [39, 59, 69]. CANHunter [87] overcomes these limitations. However, its static analysis approach relies on hard-coded vulnerabilities in vehicle-specific companion diagnostic apps, which disables it for handling scenarios where CAN configurations are dynamically retrieved from servers and limits its applicability to vehicles that do not publicly provide diagnostic apps. A detailed comparison of these approaches is presented in Section 2.3. Therefore, it is desirable to design a fully *automated* and *low-cost* CAN reverse engineering framework that *solely relies on the head unit.*

To tackle **RQ1**, we analyze the standalone head units from three vehicle models (i.e., the BYD Qin Plus, BYD Song Plus, and Lynk&Co 06 Remix), which are publicly purchasable from vehicle maintenance shops. Moreover, we also examine the widely deployed AAOS [43]. We observe that the head unit is directly connected to the in-vehicle network and achieves functionality by sending CAN messages. Although domain isolation is implemented, a vehicle control CAN message triggered by apps with system-level permissions can pass through the central gateway and domain controllers to reach in-vehicle components in other domains. Based on that, we propose IVIHunter to address **RQ2**. IVIHunter aims to utilize backward program slicing to locate vehicle control components in apps and automatically trigger them with high efficiency and coverage, through which IVIHunter can parse CAN messages sent from the head unit and correlate them to corresponding vehicle control actions through a temporal pattern-based match.

To achieve this analysis, we propose several novel designs to overcome the technical challenges. For program slicing, the effectiveness relies heavily on sink API selection, but the closed-source nature of target head unit OSs renders the vehicle control APIs unknown. Even when APIs for a specific model are identified, manufacturer-specific variations hinder transferability. To address this difficulty, IVIHunter leverages the fact that vehicle control functions are intrinsically bound to UI components and therefore treats Android native UI event listeners (e.g., click) as sink APIs. For automated triggering, evaluation reveals that GUI testers and LLM agents perform poorly on in-vehicle apps (detailed in Section 4.3.1). We overcome this by designing multi-priority rules to achieve superior coverage and efficiency.

For head units of three vehicle models and open-source AAOS, by analyzing seven built-in vehicle control apps, IVIHunter reverse-engineers 629 CAN messages corresponding to 230 vehicle control functions. Notably, some functions, like energy recycle, are classified as safety-critical by SAE J3061. Moreover, the functions of BYD vehicles span all five domains shown in Figure 1, and cross three domains in the case of Lynk&Co, forming the following attacks.

Real-world attacks driven by IVIHunter. Based on IVIHunter, we propose two real-world attacks. (1) *Remote vehicle status monitoring:* This attack exploits the system-privileged app's vulnerability to gain cross-domain access with zero permission, through which the attacker can hijack in-vehicle CAN traffic remotely and interpret it by applying IVIHunter's reverse engineering results, ultimately

Table 1: Related work of CAN reverse engineering tools.

Analysis Tool	Hardware Device Required		Semantic Identify Accuracy	Without Domain Knowledge	Specific Vulnerability Unnecessary
	Real Vehicle Free	Specialized Device Free			
READ [59]	×	✓	×	×	✓
LibreCAN [69]	×	✓	×	×	✓
AutoCAN [39]	×	✓	×	×	✓
CANMatch [19]	×	✓	×	✓	×
DP-Reverser [90]	×	×	✓	✓	✓
CANHunter [87]	✓	✓	✓	✓	×
IVIHunter	✓	✓	✓	✓	✓

achieving vehicle status monitoring. Real-vehicle experiments on a BYD Qin Plus demonstrate that 26 types of vehicle status can be monitored remotely in real time, which may undermine the driver's financial interests. For example, some insurance companies leverage driving behavior inferred by driving status to adjust users' insurance premiums in a targeted manner [79]. (2) *Malicious vehicle control command injection:* By utilizing the program analysis capability, IVIHunter performs forward control flow searches on the app's ADAS module to identify system hidden application programming interfaces (APIs) and exploits them via code reflection mechanism, enabling the malicious vehicle control. Evaluation shows that the attack can achieve control for 10 ADAS functions (e.g., deactivating traffic sign recognition), posing a significant threat to driving safety.

The main contributions of this paper are as follows:

- *Revealing cross-domain risks of the head unit.* We investigate the vehicle control capability and associated security issues within the head unit, uncovering the risk of cross-domain attacks.
- *Demystifying proprietary protocols via automated analysis.* We design IVIHunter to perform automated CAN reverse engineering, and evaluation demonstrates its high efficiency and coverage.
- *Identifying real-world attacks.* Driven by IVIHunter, we reveal the remote monitoring and vehicle control attacks, which substantially undermine driving privacy and safety.

Ethical considerations. We reported analysis results to BYD, Lynk&Co, and Google, while vulnerabilities within BYD vehicles are officially confirmed and patched. Moreover, we publicly disclosed the vulnerabilities, receiving `CVE-2025-28169` and `CVE-2024-54728`. The artifacts are available at https://github.com/rainymoods/IVIHunter.

2 Preliminaries

2.1 In-Vehicle Infotainment System

The IVI system refers to a suite of software and hardware to provide information and entertainment functions. A key component among them is the *head unit*, which resembles a tablet and serves as an essential interface for drivers. Initially, head units were limited to multimedia features. Today, they allow users to configure various vehicle settings, significantly enhancing the vehicle's intelligence.

The OSs used in head units are often custom-developed based on mature OSs to inherit their well-established architectural designs, extensive application ecosystems, and robust security mechanisms. For example, key automakers like Volkswagen, General Motors, Hyundai, BYD, Ford, Nissan, Honda, BMW, Geely, Volvo, and Lynk&Co have adopted Android-based OSs on multiple models [3, 14, 17, 22, 27, 29, 37, 46, 80, 85, 94].

2.2 Vehicle E/E and Security Architecture

Numerous attacks [26, 28, 54, 61, 62, 66, 77] targeting smart vehicles that undermine privacy and safety have been proposed. To mitigate them, security standards like ISO/SAE 21434 [49] and SAE J3061 [1] require a domain-centralized architecture, where in-vehicle components are organized into isolated domains that communicate via the CAN, with each domain coordinated by a domain controller and inter-domain communication managed by a central gateway. For example, according to the official repair manual [25], the BYD Qin Plus model employs a domain structure illustrated in Figure 1: intelligent access (IA), energy, chassis, body, and engine control module (ECM).

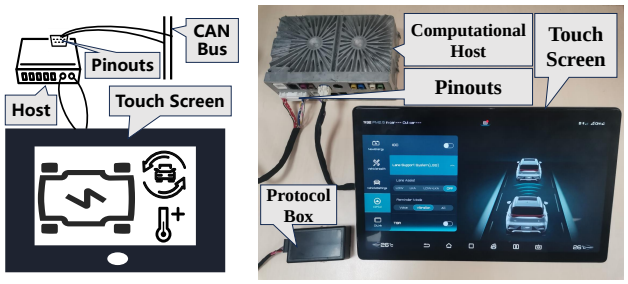
To mitigate attacker-controlled components from performing malicious cross-domain vehicle control, cross-domain control capabilities are strictly constrained by the central gateway, domain controllers, and inside the in-vehicle components. The architecture is widely adopted by major automotive manufacturers, including Volkswagen, Toyota, General Motors, Hyundai, BYD, Ford, Honda, BMW, Volvo, and Lynk&Co [23, 38, 74]. In addition, the market size of domain control units reaches $6.76 billion in 2025 and is expected to increase to $14.29 billion by 2034 [38].

2.3 CAN Protocol and Reverse Engineering

In modern vehicles, hundreds of electronic control units (ECUs) communicate through the in-vehicle network to coordinate driving functions. The CAN [2] is the most widely adopted protocol, and communication between ECUs is conducted through CAN messages, which primarily consist of an Identifier (ID) and a Data segment. The CAN ID, which is 11 bits (standard frame) or 29 bits (extended frame), indicates the message type and is also used in arbitration to resolve conflicts. The Data segment, typically 8 bytes, carries vehicle control parameters. When addressing larger payloads, CAN FD (CAN Flexible Data-Rate) supports up to 64 bytes.

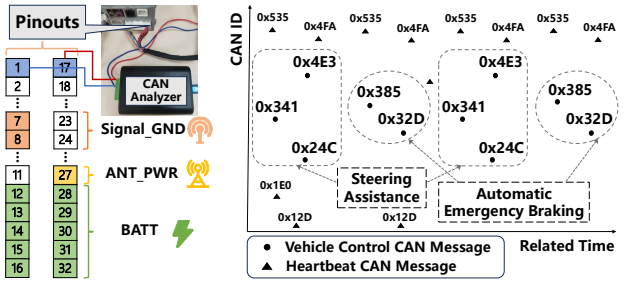
Although the CAN message format is regulated, the semantic information of its payload varies across different vehicle brands and models, and is typically regarded as proprietary. Therefore, various CAN reverse engineering tools have been proposed, while accompanied by limitations summarized in Table 1. Traffic analysis approaches based on the syntactic features of CAN messages have been widely studied [19, 39, 59, 69]. Among them, to infer semantics, READ [59] relies on expert domain knowledge, for example, by empirically judging whether the variation trend of CAN Data with the same CAN ID matches changes in vehicle speed. Similarly, LibreCAN [69] computes similarity between CAN Data trends and vehicle states collected during driving, like speed and acceleration, to perform matching. AutoCAN [39] identifies semantics by examining whether relationships among signal trends conform to physical laws, for example, those governing speed, acceleration, and traveled distance. A common limitation of these methods is that they can only provide coarse semantic inference with limited accuracy. Moreover, some manufacturers apply numerical transformations to CAN Data, causing them to deviate from original values and rendering these trend-matching-based schemes suboptimal [90].

To avoid reliance on domain knowledge, CANMatch [19] utilizes a database composed of numerous Database CAN (DBC) files to fingerprint unknown CAN messages against those in the database



(a) General structure. (b) BYD Qin Plus structure.

Figure 2: An illustration of the head unit's structure.



(a) Host pinouts. (b) In-vehicle CAN traffic.

Figure 3: Extract CAN messages from pinouts of head unit.

for decoding. However, it relies on the reuse of CAN format across vehicles, which is not common among vehicle brands, and the used commercial database is not publicly accessible. DP-Reverser [90] employs a robotic arm to eliminate manual efforts. However, it requires costly professional diagnostic devices and exceeds the standard attacker capability defined in ISO/SAE 21434 [49].

All the above methods depend on real vehicles or manual driving for data collection, which makes their deployment costly and labor-intensive. CANHunter [87] avoids these limitations by conducting static code analysis on companion diagnostic apps. However, it relies on hard-coded CAN message vulnerabilities in vehicle-specific apps, thus failing to generalize to nowadays apps where CAN configurations are dynamically retrieved from servers, or to vehicle models that do not publicly provide companion diagnostic apps.

3 Insights and Problem Formulation

3.1 Insights from Dissecting Head Unit

IVIHunter aims to utilize the head unit as an analysis interface to demystify in-vehicle protocols and facilitate cross-domain attacks. This attack path does not require physical access to the victim's head unit, but leverages the fact that the same vehicle models typically share identical CAN instruction sets [19]. This allows attackers to obtain head units of the same model as the target vehicle and perform CAN reverse engineering solely on them using IVIHunter, while applying the derived results to launch real-world attacks against victim vehicles. The research insights can be refined into the following questions:

- **Q1:** What are the underlying principles that enable the head unit to connect to or control in-vehicle components in other in-vehicle domains?
- **Q2:** Considering the limitations of previous approaches and their mismatch with our analysis assumptions (i.e., presented in Table 1), is it feasible to design an automated CAN reverse engineering tool with high efficiency and low cost?
- **Q3:** If the analysis interface exists, is it limited to a single vehicle model or holds across different models and manufacturers?
- **Q4:** Driven by IVIHunter, is it possible to launch real-world cross-domain attacks that threaten driving privacy and safety?

To address the research questions **Q1** and **Q2**, we present several case studies conducted on the head unit of a BYD Qin Plus, which operates on DiLink OS 3.0 [24] and employs a domain-centralized E/E architecture. For **Q3**, we further analyze the head units on BYD Song Plus and Lynk&Co 06, and we also examine the AAOS 12L, since 51 vehicle models from manufacturers such as Volkswagen, GM, Hyundai, and Ford are using or planning to adopt AAOS [27, 46, 88]. Finally, driven by IVIHunter, we take a further investigation of BYD Qin to reveal potential real-world attacks to answer **Q4**.

3.1.1 Dessecting a Standalone Head Unit. To resolve **Q1**, we analyze the head unit of BYD Qin Plus. As shown in Figure 2, it consists of a touchscreen and a computational host, running the DiLink OS. Moreover, the head unit is connected to the CAN bus and located within the intelligent access domain, providing vehicle control functions through cross-domain CAN communication. Since the CAN protocol does not incorporate encryption mechanisms, we sniff CAN messages by connecting a commercial CAN analyzer (i.e., USBCANFD with a price less than $50) to the CAN bus pinouts of the computational host, as illustrated in Figure 3(a).

We summarize the workflow of the head unit. Taking the triggering of `steering assistance` as an example, when the user clicks the switch on the touchscreen, the event is received by the computational host and processed by the DiLink OS. DiLink converts the triggered vehicle control function into corresponding CAN messages, which are sent to the intelligent access domain via CAN pinouts of the computational host. We observe that each action has a unique and stable traffic pattern, such as messages with CAN ID 0x4E3, 0x341, and 0x24C for steering assistance, with the automatic emergency braking function exhibiting the same characteristic, as shown in Figure 3(b). These messages are then inspected by the central gateway's firewall and routed to the chassis domain [25], executing requested configuration through related devices.

However, *this capability is only available to system-privileged apps.* Requests from regular apps are rejected by DiLink. Additionally, this cross-domain access capability is unique to the head unit. For instance, when an attacker compromises a component in the body domain and sends a CAN message to turn off the engine, the message would be blocked by the central gateway.

Observation 1: *The head unit processes cross-domain control ability, provided the command is generated by system-privileged apps.*

3.1.2 Exploration on Automated Analysis. To perform CAN reverse engineering, we need to parse the CAN messages corresponding to vehicle control functions. Based on **Observation 1**, this can be achieved by systematically triggering functions within head unit apps while monitoring the resulting CAN traffic. However, designing IVIHunter to automate this process presents several challenges.

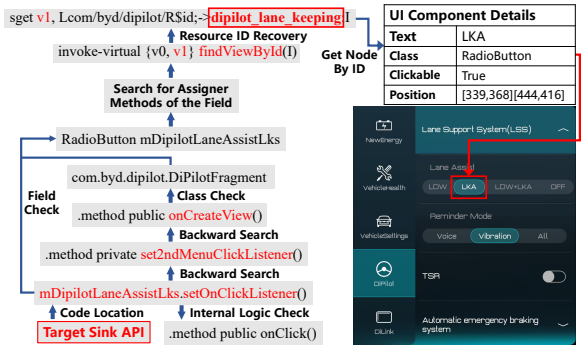


Figure 4: Locate vehicle control components by code analysis.

The initial difficulty lies in identifying vehicle control functions with high coverage and accuracy. Existing methods exhibit significant inadequacies in head unit context: for instance, visual-based methods (e.g., DP-Reverser [90]) utilize optical character recognition (OCR), which is inherently designed for text extraction and thus incapable of interpreting domain-specific icons prevalent in automotive UIs. Additionally, CANHunter [87] adopts code analysis but relies on the hard-coded CAN data vulnerability in apps, which is inapplicable to our scenario and uncommon in modern IVI OSs. Another challenge arises from efficiently triggering all components, which necessitates a finely optimized execution sequence. Although LLM agent and GUI tester are potential solutions, their limited visual understanding [78] and illogical interaction strategies lead to poor performance in the IVI setting, as evaluated in Section 4.3.1.

To address the identification challenge, IVIHunter leverages the fact that detailed information about vehicle control components is defined in the app program. Although DiLink is a closed-source system, it is, like most mainstream head unit OSs, developed based on the native open-source Android platform [17, 22, 29, 37], and thereby inherits the Android framework, which facilitates the analysis. Based on this insight, IVIHunter performs backward program slicing analysis (to be discussed in Section 4.2) on apps to extract vehicle control functions UI components (as presented in Figure 4). For the triggering phase, Android's native software service is used to simulate human-like clicks on identified components, and the multi-priority triggering rule is designed to trigger more components in less time (explained in Section 4.3).

Ultimately, by leveraging time pattern-based matching, we identify the correspondence between vehicle control actions and CAN messages, motivating the subsequent observation for **Q2**.

Observation 2: *Efficient and automated CAN reverse engineering can be achieved purely through the head unit analysis.*

3.1.3 Investigation of the Generalizability across Different Vehicles. To reflect the diversity of the current automotive market, we categorize head unit OSs and conduct analysis on the representative device in each category. At the OS level, we analyze both native in-vehicle OSs (i.e., the AAOS 12L) and customized systems built upon native OSs (i.e., head unit OS of the Lynk&Co 06 Remix, running LYNK OS L based on AAOS 9). Moreover, to represent cross-model transferability within the same manufacturer, in addition to the BYD Qin Plus, we further analyze the BYD Song Plus, which runs DiLink 4.0 OS built on Android 10.

For **Observation 1**, we find that the BYD Song Plus also adopts a domain-centralized architecture primarily divided into IA, energy, chassis, body, and ECM domains. Moreover, the Lynk&Co 06 employs a comparable strategy, with corresponding domains including information, hybrid, chassis, comfort, and powertrain. When performing cross-domain vehicle control actions, CAN messages are initiated by the head unit and routed through the central gateway to target ECUs. Likewise, in all these head units, such cross-domain capabilities require system-level privileges. As the AAOS runs in a virtual machine, it does not correspond to an in-vehicle network architecture before being deployed on a real vehicle. However, as described in the official document, its communication and permission design principles align with the findings mentioned above [4–6].

For **Observation 2**, all head units we analyzed are developed based on Android, which allows applying the automated vehicle control actions triggering method to them. Additionally, both BYD and Lynk&Co models adopt the CAN series protocols and send messages from pinouts, enabling the capture and parsing of CAN traffic. For AAOS, we conduct development based on the open-source code according to the official specifications [5, 44], define a CAN instruction set, and collect the CAN traffic through abstract hardware. Therefore, we draw the ensuing conclusions to **Q3**.

Observation 3: *The domain-centralized architecture and permission-based cross-domain capability management exhibit generalizability among different vehicle models, and the automated analysis method also shows universality.*

3.1.4 Exploration of Cross-domain Attacks. Although CAN reverse engineering results can be widely applied [87], like vehicle diagnosis and autonomous driving development, we aim to address **Q4** by revealing cross-domain attacks threatening privacy and safety.

Vehicle status information is explicitly reflected in in-vehicle CAN communication traffic. Based on the CAN reverse engineering results of IVIHunter, the vehicle status can be inferred through observing the presence of specific CAN messages in the traffic. However, a key challenge arises in that these cross-domain CAN traffic collection capabilities are exclusively granted to system-privileged apps. To address the issue, IVIHunter performs backward program slicing on the system CAN collection app to analyze the data flow of CAN data from its collection to processing. It is observed that the head unit of BYD Qin frequently (e.g., per minute) uploads CAN traffic to a cloud server and that the destination address is configured through an unprotected broadcast. This design allows a zero-permission app to modify the upload address to an attacker-controlled server by sending such a malicious broadcast, thereby enabling the remote vehicle status monitoring attack.

To compromise safety by launching malicious vehicle control, many prior attacks have shown that even after obtaining privileged access in head unit through vulnerabilities, command injection still requires complex manual analysis [26, 36, 50, 55, 62, 66, 76], such as reverse engineering the OS, understanding hardware drivers, and parsing communication protocols. To avoid such effort, IVIHunter leverages the cross-domain capabilities of head unit and performs a forward control flow analysis on backward program slicing results to identify hidden vehicle control APIs. Since the closed-source nature of the OS prevents exploitation of these APIs through the

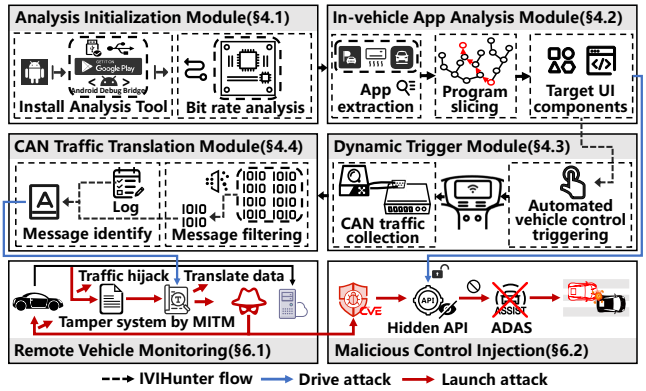


Figure 5: IVIHunter's workflow and the driven attacks.

official software development kit (SDK), we overcome this by using the reflection mechanism. As a result, malicious control attacks are enabled, and manual tasks are significantly reduced by IVIHunter.

Based on these studies, the following observation is concluded.

Observation 4: *Driven by IVIHunter, remote vehicle status monitoring and malicious vehicle control command injection attacks can be realized on specific vehicle models.*

3.2 Goals and Assumptions of IVIHunter

In this subsection, we present the goals and assumptions of IVIHunter, while the threat models and procedures of real-world attacks driven by IVIHunter are discussed in Section 6.

Analyze Goals. Based on the four observations outlined in Section 3.1, our goal is to analyze the in-vehicle network (i.e., performing CAN reverse engineering) from the head unit's perspective. This includes uncovering both syntax and semantics of critical vehicle control actions, meaning that IVIHunter should extract the manufacturer-defined corresponding CAN messages or, for certain CAN messages, infer their meanings. Finally, the CAN reverse engineering results, together with the program analysis results, will be utilized to launch real-world attacks to evaluate the threats posed by IVIHunter, and the details will be demonstrated in Section 6.

Assumptions. IVIHunter involves analysis across multiple aspects. At the hardware level, we assume that access is restricted to the standalone head unit obtained by the attacker, such as through purchasing from repair shops, secondary markets, or downloading official public system image emulators (e.g., Volvo XC40 and EX30 [86]), and that it leverages only the functionalities it provides.

From a software perspective, with the increasing openness of head units and the attacker's possession of the head unit, we assume that we can install an analysis app on the head unit that only requires normal permissions, which can be achieved through multiple approaches. A fundamental method is to transfer the analysis app to the OS via a USB flash drive [21, 30, 75], or to upload it online and download it onto the head unit through preinstalled software [41]. Moreover, the analysis app can be distributed through the head unit app store. For example, DiLink allows developers to publish apps via its official app store [20], while the AAOS-based OSs integrate the Google Play app store [31, 70, 72] and support in-vehicle app development and submission [35]. Since the analysis app is designed for in-vehicle app extraction and automated vehicle control function triggering, neither of which are typical malware features, it can potentially pass app store vetting. For instance, our evaluation using VirusTotal [83] demonstrates that none of the 67 leading global malware detection engines flagged the analysis app as malicious. In addition, debugging modes and even privilege escalation access have been revealed in numerous vehicle models [26, 36, 40, 50, 55, 76], which can also be leveraged to install the analysis app. Furthermore, we assume that communication between the head unit and the in-vehicle network applies the CAN series protocol, and that the in-vehicle apps are not obfuscated. To justify the latter, we conduct the analysis of 174 AAOS apps collected from Google Play, which reveals that only two apps (1.15%) employ control flow obfuscation, indicating that such protection remains underutilized in the in-vehicle app ecosystem.

4 System Design of IVIHunter

The workflow of IVIHunter and driven attacks is shown in Figure 5. The Analysis Initialization Module configures the environment. And then the In-vehicle App Analysis Module extracts vehicle control UI components through program analysis. The automated triggering of these components is handled by Dynamic Triggering Module. Finally, the CAN Traffic Translation Module parses corresponding CAN messages for vehicle control actions from the traffic. Driven by IVIHunter, further cross-domain attacks are proposed.

4.1 Analysis Initialization of IVIHunter

4.1.1 Installing the Analysis Tool. To extract vehicle control apps from the head unit for program analysis, and to execute subsequent automated triggering tasks, IVIHunter installs an analysis tool on the OS (i.e., a normal app without any system-level permission). Since vehicles of the same model adopt identical CAN formats, it is feasible to install the app on our own head unit for analysis, and the obtained results can be directly reused in the following attacks.

As described in Section 3.2, this can be achieved through multiple approaches according to availability. For example, for BYD vehicles, a USB flash drive can be used as the transfer medium. Moreover, the ADB service is accessible in the Lynk&Co 06, which enables installation via a laptop connected through a USB cable. As to AAOS, since it provides a built-in web browser app, the analysis app can be uploaded online and then downloaded for installation. Additionally, the app store in the OSs could also be utilized, like BYD and AAOS.

4.1.2 Sniffing the CAN Traffic. As presented in Figure 3(a), to receive CAN messages sent from the head unit, it is essential to analyze communication parameters. IVIHunter traverses common parameter combinations and observes whether CAN messages are received to determine validity. IVIHunter reveals that, for BYD Qin and Song, the CAN FD protocol is adopted, with the arbitration field baud rate set to 500 kbps, and the data field baud rate configured to 2 Mbps. For Lynk&Co 06, the CAN protocol is applied, with arbitration and data field baud rates both set to 500 kbps.

4.2 In-vehicle App Analysis

In this subsection, IVIHunter extracts system-privileged vehicle control apps and performs backward program slicing on them. Through this process, IVIHunter obtains details about the vehicle control UI components within the apps.

Table 2: Part result of in-vehicle app extraction and vehicle control UI component identification.

IVI System	Vehicle Control App	Listener Function	Common Component Type	View Changes after Triggering			UI Components #	Vehicle Control Functions #
				Slight	Partial	Window		
DiLink 3.0	Airconditioning	setOnClickListener	ImageView, Button…	44	7	1	27	33
		setOnTouchListener	Button, Switch…				25	
	Carsettings	setOnClickListener	ImageView, Button…	104	148	20	152	61
		setOnCheckedChangeListener	RadioGroup…				76	
		setOnSeekBarChangeListener	SeekBar…				10	
DiLink 4.0	Airconditioning	setOnClickListener	ImageView, Button…	45	8	1	28	34
		setOnTouchListener	Button, Switch…				26	
	Carsettings	setOnClickListener	ImageView, Button…	99	19	2	113	39
		setOnCheckedChangeListener	RadioGroup…				3	
		setOnTouchListener	Button, Switch…				2	
LYNK OS L	Air Conditioner	setOnClickListener	ImageView, Button…	19	47	-	60	22
		setOnTouchListener	Button, Switch…				3	
		addUpdateListener	ProgressBar…				1	
	My Car	setOnCheckedChangeListener	RadioGroup…	67	23	-	50	23
		setOnClickListener	ImageView, Button…				37	
AAOS 12L	System UI	setOnClickListener	ImageView, Button…	129	-	-	95	20
		setOnTouchListener	Button, Switch…				12	
		addUpdateListener	ProgressBar…				12	

4.2.1 Vehicle Control App Scanning. To identify target vehicle control apps, IVIHunter examines the app attributes provided by the system package manager. For example, IVIHunter collects package information (e.g., the package name) and app metadata (e.g., the app label), and inspects the domain-specific keywords in them, such as `carsetting` and `vehicle control`, to recognize candidate apps.

IVIHunter identified seven vehicle control apps described in Table 2. In terms of DiLink 3.0, the `AirConditioning` app and the `CarSettings` app are detected, which integrate nearly all vehicle control-related functions provided by DiLink. The functionalities within the `AirConditioning` app are divided into three modules: air conditioning settings, seat ventilation and heating, and air purification, while the `CarSettings` app supports five categories of functions: new energy power configuration, ADAS, body and cabin settings, vehicle status maintenance, and system settings.

4.2.2 Vehicle Control UI Component Identification. To enable the subsequent automated triggering described in Section 4.3, an essential task is to extract all vehicle control UI components and their associated information. IVIHunter achieves this through the following backward program slicing analysis.

Top function localization via control flow analysis. To perform program analysis, it is observed that clicking a UI component triggers the vehicle control action because the component is bound to relevant *listener functions* (e.g., `setOnClickListener`, etc.), thereby invoking vehicle control logic within them. Moreover, since even third-party listener functions need to call official APIs to implement functionality, the observation generalizes to other Android-based systems. Although this property allows us to locate components by identifying listener functions, such bindings involve complex interprocedural call chains spanning multiple classes. To address this challenge, IVIHunter performs backward program slicing, which first constructs the app control flow graph and then treats official Android listener binding APIs as analysis entry points. By reverse tracing functions that invoke these APIs recursively, IVIHunter identifies top-level functions (e.g., `initView` or `setupUI` functions).

Vehicle control UI components association. The following obstacle is how to further associate top functions with the vehicle control UI components. IVIHunter leverages that when a component class object is created, its corresponding class constructor function is called, allowing us to locate UI components from the top constructor function or identify them from layout files according to the class type. However, this method applies only to cases where the listener function is bound within the constructor function. For failure cases where listeners are bound outside (such as through `onCreate` or `onCreateView` functions), IVIHunter focuses on classes defining the top functions, recognizes potential vehicle control UI components by checking whether their view field objects are assigned and bound to listener functions through the top functions.

Analysis results. The analysis results are shown in Table 2. Take DiLink 3.0 as an example, IVIHunter extracts 52 and 272 suspected vehicle control UI components from the `AirConditioning` app and the `CarSettings` app. These components serve the needs of different vehicle control functions. For example, the air conditioning switch may use the `Button` type, the electricity storage value employs the `SeekBar` type, and the lane assistance, which features four sub-modes, is well-suited for the `RadioGroup`.

4.2.3 Component-related Information Extraction. To support the following automated triggering, IVIHunter extracts component-related information, including resource identifiers, view changes caused upon triggering, and vehicle control semantics.

Extraction of component resource identifiers. During automated triggering, resource identifiers are used to locate vehicle control UI components shown on the screen and perform human-like interactions on them (e.g., click). Since resource identifiers are typically served as a parameter of the `findViewById` function to be assigned to UI components, IVIHunter records all assignment functions associated with view type field objects when constructing the control flow graph, and extracts the resource identifiers by analyzing their internal logic.

Collection of additional component information. The view changes caused by triggering UI components will be used to set priority during automated triggering. These changes are categorized by the invoked APIs upon triggering, such as `commit` for partial transitions, `startActivity` for window switches, and no such APIs for slight

Table 3: Vehicle control app analysis results and the correctness verification of CAN reverse engineering.

The Head Unit OS	Vehicle Control App	Extracted Components #	Recognized as CAN-related #	Actual CAN-related #	Correctly Extracted #	Accuracy	Coverage	Precision	F1 Score
DiLink 3.0	AirConditioning	52	34	33	33	98.08%	100%	97.06%	98.51%
	CarSettings	272	63	61	61	99.26%	100%	96.83%	98.38%
DiLink 4.0	AirConditioning	54	33	34	33	98.15%	97.06%	100%	98.50%
	CarSettings	120	41	39	39	98.33%	100%	95.12%	97.50%
LYNK OS L	Air Conditioner	66	21	22	21	98.48%	95.45%	100%	97.67%
	My Car	90	23	23	23	100%	100%	100%	100%
AAOS 12L	System UI	129	21	20	20	99.22%	100%	97.73%	98.85%

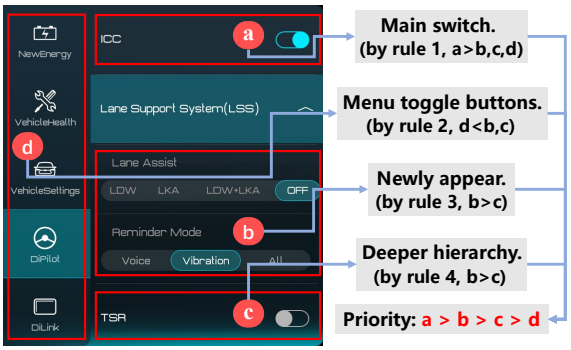


Figure 6: Triggering priority determined by the algorithm.

changes. By treating these APIs as entry functions and applying backward program slicing, the view changes induced by activating components are identified. For vehicle control semantic analysis, resource identifiers, function call chains, and component textual contents are jointly analyzed by methods like keyword matching. Furthermore, this analysis is used to determine whether a component carries specific logical semantics like the main control switch based on component type (e.g., button-related), resource identifier (e.g., contains "switch" field), and textual information.

Performance analysis. Here, the view change information illustrated in Table 2 is used as an example, in which `Slight` indicates no significant change, such as the lighting effect when a button is pressed. `Partial` refers to substantial changes, like the transition between different modules, while `Window` involves a complete window switch. As exemplified by DiLink 3.0 analysis results, for the `AirConditioning` app, the counts for these categories are 44, 7, and 1, while for the `CarSettings` app, they are 104, 148, and 20.

4.3 Dynamic Command Triggering

This module automates the triggering of extracted vehicle control components to generate the corresponding CAN messages.

4.3.1 Automated Triggering Based on Multi-priority Rule. The automated triggering is conducted by the analysis tool in Section 4.1.1 based on program analysis results. To identify CAN messages within noisy traffic, this process adheres to specific temporal patterns, such as predefined click sequences. Moreover, to enhance coverage (triggering more functions) and efficiency, it is essential to optimize the operation sequence. Since the LLM agent and GUI tester exhibit limitations in the head unit scenario, such as difficulty in recognizing app icons [78] and reliance on illogical interaction strategies, we propose a multiple-priority rule triggering algorithm.

Specifically, as excessive granularity may lead to rule conflicts while overly coarse rules diminish optimization effectiveness, we strike a balance by defining priority rules across four dimensions: task logic, contextual relevance, time, and space. IVIHunter dynamically assigns priority to visible and untriggered UI components and adds them to a priority tree. Each time, IVIHunter pops the highest-priority component and triggers it, while adding the newly appeared ones. These rules are described below in order of priority.

Rule 1: Components with special logical meanings should be prioritized for triggering. Since the availability of some functions relies on whether certain components are activated, buttons with logical semantics are prioritized. For instance, to configure the lane support system shown in Figure 6, a precondition is that the main control switch of the intelligent cruise control (ICC) system is enabled.

Rule 2: Buttons that cause minimal changes to the view should be prioritized. Major view changes triggered by a component disrupt the context and should be delayed. For example, if the main menu button on the far left in Figure 6 is clicked, the entire right view will change. Therefore, all functions in the current window should be executed before clicking the menu switch to prevent any missing.

Rule 3: The most recently appeared buttons should be triggered first. The time feature is considered because clicking some components results in the appearance of new vehicle control functions. In such cases, the newly displayed components should be prioritized, since triggering others first may cause them to disappear and be missed. As presented in Figure 6, the newly appeared lane assistance function should be triggered before the traffic sign recognition (TSR).

Rule 4: Buttons located at the deeper view hierarchy should be prioritized. The analysis of view space characteristics reveals that, since the hierarchy is structured as a tree, a deeper node signifies finer-grained functional units and should be prioritized. For instance, all parameters of a function should be configured before clicking the confirmation button. As illustrated in Figure 6, the lane assistance occupies a deeper hierarchy level than TSR.

Evaluation results. Take the evaluation on DiLink 3.0 as an example, all 52 components of the `AirConditioning` app are triggered in 90.508s, while all 272 components of the `CarSettings` app are triggered in 357.609s, both achieving 100% coverage. Furthermore, the test coverage for apps in the remaining 3 head unit OSs all exceeded 95%. For detailed results, the coverage test is shown in Table 3, where coverage is defined as the percentage of vehicle control components that are successfully triggered. And efficiency results are summarized in Table 7 of Section 5.2.

Comparison with alternative methods. To demonstrate the advantages, we compare IVIHunter with other alternative methods by testing on `AirConditioning` app, with results presented in Table 4.

Manual Triggering. In our experiment, we manually click vehicle control components and observe the corresponding CAN messages. However, due to the high complexity of CAN traffic, this approach

Table 4: Different methods to perform automated triggering.

The Method	Time Cost per Function(s)	Recognition Rate	Trigger Coverage	CAN Message Accuracy
Manually	129.13	100%	100%	36.36%
LLM Agent	18.82	57.58%	18.19%	33.33%
GUI Tester	\	60.61%	\	\
IVIHunter	2.74	100%	100%	98.08%

is highly time-consuming and labor-intensive, and it frequently fails to capture relevant CAN messages, resulting in low accuracy.

LLM Agent. We employ AppAgent [92], an advanced LLM agent specifically designed for Android GUI interaction, for evaluation. The results show that it identifies only 19 vehicle control components and triggers 6 of them. This failure arises because AppAgent misjudges certain actions as unhelpful to the task and attempts to revert them using back commands, which inadvertently terminate the app. Moreover, AppAgent falls short in visual component understanding, as the in-vehicle apps often use small icons to indicate the vehicle control functions, which AppAgent struggles to interpret.

GUI Tester. UI Automator [7] is an official Android app GUI testing framework that identifies component attributes (e.g., *clickable*) to determine supported operations. However, it exhibits suboptimal accuracy. During evaluation on `AirConditioning`, the on/off button is incorrectly labeled as non-clickable, causing it to be ignored. More critically, when the air conditioning system is not activated, other functions remain unavailable, rendering subsequent operations ineffective. Therefore, although 60.61% components are accurately recognized, none of the vehicle control actions are triggered.

In summary, the results show that IVIHunter is 47.65× more efficient than manual operation, achieves 81.81% higher trigger coverage than an LLM agent, and improves component recognition by about 40% over the GUI tester, which highlights its effectiveness.

4.3.2 Vehicle Control Function Triggering Logging. To recognize CAN messages within the complex traffic based on specific temporal patterns followed during triggering, IVIHunter records the logs, which include operational details such as the total number of clicks, time intervals, timestamps, and the structure of clicked buttons (e.g., sub-button numbers). Additionally, IVIHunter documents the details of the triggered components, such as resource identifiers, component text, and content description information, which collectively reflect the semantics of the extracted CAN messages.

4.4 CAN Traffic Translation

4.4.1 Traffic Pre-processing. The CAN bus contains high-frequency heartbeat messages that disrupt analysis. To mitigate it, IVIHunter sniffs CAN traffic and counts messages by CAN ID to compute a minimum frequency threshold. During filtering, messages exceeding this threshold are classified as heartbeat messages and excluded.

4.4.2 Temporal Pattern-based CAN Message Extraction. The goal of this step is to combine trigger logs with filtered CAN traffic to identify CAN messages of vehicle control functions.

Prerequisites analysis. Since the automated triggering follows a specific temporal pattern, the resulting CAN messages exhibit corresponding characteristics. Based on the experiment, the time difference between clicking a vehicle control component and the corresponding CAN messages being captured is approximately 1 ms. Additionally, during a single trigger event, the head unit typically sends multiple repetitive CAN messages to ensure robustness. These properties enable the IVIHunter to achieve high accuracy.

Temporal pattern-based CAN message extraction. IVIHunter extracts CAN messages corresponding to a vehicle control function in CAN traffic by calculating the time boundary. This process begins by synchronizing the timestamp of the traffic with the trigger log at the starting point, which serves as the left boundary for searching, since CAN messages are not sent before the vehicle control is triggered. It then gets the time interval until the next trigger event from the log. This interval, together with a threshold(e.g., half of the click interval), defines the right boundary. Using this determined range, IVIHunter searches the corresponding time segment in the CAN traffic to extract CAN messages. Since the trigger logs record the functional meaning of the executed vehicle control functions, the semantics of the CAN messages can be identified. Therefore, these steps complete the CAN message extraction process for a single vehicle control action, and repeating them accomplishes the task.

Table 5: Overall result of IVIHunter on different domains.

IVI System	Vehicle Control App	Relevant Domain	Vehicle Control Functions #	CAN Message #
DiLink 3.0	CarSettings	IA	29	71
		Energy	6	15
		Chassis	3	7
		Body	20	44
		ECM	3	5
	AirConditioning	IA	29	84
		Body	4	10
DiLink 4.0	CarSettings	IA	19	48
		Energy	2	5
		Chassis	1	3
		Body	15	38
		ECM	2	4
	AirConditioning	IA	31	91
		Body	2	5
LYNK OS L	My Car	Information	3	11
		Chassis	11	28
		Comfort	9	36
	Air Conditioner	Comfort	21	58

5 Evaluation of IVIHunter

In this section, we show the reverse engineering results of IVIHunter on three vehicle modes and AAOS.

5.1 Experiment Environment Setup

In our experiments, the head units are salvaged units from decommissioned vehicles. In-vehicle app analysis (described in Section 4.2) is performed on a desktop computer, configured with an Intel i7-14700KF CPU and 32GB of RAM. The CAN analyzer supports the standard CAN and CAN FD protocols. These devices are defined in ISO/SAE 21434 as the lowest-level (i.e., Standard) attack equipment.

5.2 CAN Reverse Engineering Results

CAN message characteristics analysis. IVIHunter achieves CAN reverse engineering with results shown in Table 5. For example, in DiLink 3.0, all 33 vehicle control functions for `AirConditioning` app and all 61 in `CarSettings` app are identified, corresponding to 94 and 142 CAN messages. These messages involve access to

Table 6: Sample CAN reversing engineering results of BYD Qin, BYD Song, and Lynk&Co 06 models.

Head Unit	Vehicle Control App	Relevant Domain	Functional Category	Vehicle Control Command	CAN ID	CAN Data
DiLink 3.0	CarSettings	Intelligent Access Domain	Cockpit Setup	Electric AC Heating Mode	0x4F*	** ** ** ** C4 FF ** **
				Remote Air Conditioning Schedule	0x4D*	** ** ** ** ** ** 00 02
		Chassis Domain	Driving Mode	Comfort Steering Assistance	0x24*	00 01 ** ** ** ** ** **
					0x4E*	** ** 00 04 ** ** ** **
					0x34*	** ** ** 08 00 ** ** **
				E-Pedal Mode	0x2B*	** ** ** FF 01 ** ** **
		Energy Domain	Energy Management	Set the State of Charge value to 20%	0x4E*	** ** ** ** ** ** 19 00
					0x34*	** ** ** ** 19 00 ** **
				Vehicle External Discharging	0x2B*	40 00 ** ** ** ** ** **
		Body Domain	Body Control	Long Press of Smart Key to Lock and Close the Window	0x24*	00 01 ** ** ** ** ** **
					0x4E*	** ** 40 00 ** ** ** **
				Automatic Locking During Driving	0x4E*	** ** ** ** ** ** 80 00
		ECM Domain	Engine State	Standard Energy Recycle Mode	0x4E*	** ** 00 40 ** ** ** **
					0x34*	** ** 00 08 ** ** ** **
				Memory Sport Mode	0x2B*	** ** ** ** ** ** 00 08
	AirConditioning	Intelligent Access Domain	Cockpit Setup	Turn on the Air Conditioning	0x1D*	** ** 00 10 ** ** ** **
					0x4E*	09 80 ** ** ** ** ** **
					0x40*	** ** 69 31 ** ** ** **
				Heated Driver's Seat	0x43*	00 01 ** ** ** ** ** **
					0x4F*	18 00 ** ** ** ** ** **
					0x41*	09 00 ** ** ** ** ** **
				Lock Rear Pushbutton	0x4F*	** ** ** ** C0 FF ** **
					0x4E*	09 40 ** ** ** ** ** **
		Body Domain	Environmental Monitoring	Monitoring PM2.5	0x4F*	** ** 07 FF ** ** ** **
				Purify the Air	0x4F*	** ** 03 FF ** ** ** **
DiLink 4.0	CarSettings	Intelligent Access Domain	Cockpit Setup	Auto Air Recirculation (Parked)	0x4E*	00 0(48) ** ** ** ** ** **
				AC Energy Consumption Setting	0x4E*	(12)0 00 ** ** ** ** ** **
					0x24*	00 03 ** ** ** ** ** **
		Chassis Domain	Driving Mode	Steering Assist Mode	0x4E*	** ** 00 0(4C) ** ** ** **
					0x24*	00 0(13) ** ** ** ** ** **
				E-pedal Mode	0x2B*	** ** ** FF 0(12) ** ** **
		Energy Domain	Energy Management	Memory Forced State of Charge Hold	0x2B*	** ** ** ** ** ** 00 (48)0
		Body Domain	Body Control	Daytime Running Light	0x43*	** ** ** ** ** ** 00 (48)0
				Auto Door Lock While Driving	0x4E*	0(12) 00 ** ** ** ** ** **
					0x49*	** 01 02 ** ** ** ** **
		ECM Domain	Engine State	Memory Max EV Mode	0x2B*	** ** ** ** ** ** 00 (12)0
				Memory Sport Mode	0x2B*	** ** ** ** ** ** 00 0(48)
	AirConditioning	Intelligent Access Domain	Cockpit Setup	Driver Seat Heating	0x43*	00 20 ** ** ** ** ** **
				Max AC Cooling	0x4F*	** 00 (12)0 ** ** ** ** **
					0x4E*	09 80 ** ** ** ** ** **
				In-Cabin Ventilation	0x1D*	** ** ** ** ** ** 00 0(48)
					0x4E*	0(69) 80 ** ** ** ** ** **
		Body Domain	Environmental Monitoring	Purify the Air	0x4F*	00 0(48) ** ** ** ** ** **
					0x4E*	09 80 ** ** ** ** ** **
LYNK OS L	My Car	Information Domain	Driving Assist	Traffic Sign Recognition	0x2A*	** ** ** ** (01)0 00 ** **
				Forward Collision Warning	0x2A*	** ** 0(01) 00 ** ** ** **
				Collision Warning Sensitivity	0x2A*	** ** (123)0 00 ** ** ** **
		Chassis Domain	Active Safety	Lane Keeping Assist	0x2A*	00 0(08) ** ** ** ** ** **
				Automatic Emergency Braking	0x2A*	** ** 0(04) 00 ** ** ** **
				Lane Change Warning	0x2A*	0(08) 00 ** ** ** ** ** **
		Comfort Domain	Body Control	Auto Window Close on Vehicle Lock	0x2A*	** ** ** ** 0(0C) ** 04(0) **
				Auto Door Unlock on Engine Off	0x2A*	** ** 0(03) 7F ** ** ** **
				Trunk Unlock on Approach	0x2A*	00 (02)0 ** ** ** ** ** **
	Air Conditioner	Comfort Domain	Cockpit Setup	Air Conditioning Switch	0x2A*	** ** (01)7 1F ** ** ** **
				Front Window Defog	0x2A*	0(08) 0F ** ** ** ** ** **
				Set Driver Temperature to 21°C	0x2A*	** ** 07 09 ** ** ** **

five domains[1], as outlined in Table 6, including cockpit setup in the IA domain, steering assistance in the chassis domain, energy configuration in the energy domain, door and window in the body domain, and energy recycle in the ECM domain. The analysis results for DiLink 4.0 and LYNK OS L exhibit similar characteristics and are provided in Table 6, where the content in parentheses denotes different actions supported by a given function (e.g., on and off, various modes). For AAOS, CAN messages of vehicle control actions are defined by us during development and are primarily used for CAN traffic triggering and result validation.

[1]The CAN data in Table 6 is partially hidden to highlight critical fields that determine vehicle control semantics and to protect the carmaker's proprietary information.

Table 7: The average time cost of IVIHunter.

IVI System	Vehicle Control App	App Analysis(s)	Dynamic Trigger(s)	Traffic Translation(s)	Total(s)
DiLink 3.0	AirConditioning	6.482	90.508	0.126	97.116
	CarSettings	12.782	357.609	0.345	370.736
DiLink 4.0	AirConditioning	2.283	91.006	0.120	93.409
	CarSettings	5.945	134.422	0.142	140.509
LYNK OS L	Air Conditioner	2.473	81.933	0.123	84.529
	My Car	3.740	93.032	0.118	96.890
AAOS 12L	System UI	11.971	55.069	0.124	67.164

Time cost and results validation. The average time cost is shown in Table 7. For example, IVIHunter completes the CAN reverse engineering of vehicle control functions within the `AirConditioning` and `CarSettings` app of DiLink 3.0 in 97.116s and 370.736s. The majority of the time is spent on the Dynamic Command Triggering Module, accounting for 90.508s and 357.609s. In the experiment, we set each component to be clicked 4 times with a 200ms interval, which can be optimized to improve the efficiency of this module.

We also validate IVIHunter's accuracy through replaying CAN reverse engineering results on the real vehicle and observing whether corresponding vehicle control actions are executed. Meanwhile, the ground truth of AAOS is set by us during development, thus facilitating the result validation. Evaluations on DiLink 3.0 demonstrate that all vehicle control components within the two primary apps are identified, with a CAN message accuracy of 99.07%. For the other three OSs, both coverage and accuracy consistently exceeded 95% and 98%, respectively. Detailed results can be found in Table 3.

Vehicle control message features observation. We further analyze the result of DiLink 3.0 described in Table 6, and summarize the following characteristics. First, the number of CAN messages corresponding to each vehicle control action varies. For instance, turning on the air conditioning involves 3 CAN messages, while the E-Pedal driving mode only requires 1. Additionally, we discover that different vehicle control functions may use messages with the same CAN ID. For example, both turning on the air conditioning and locking the rear pushbutton use the CAN ID 0x4E∗. Finally, for the same function, opposing commands are often represented by altering fixed position values in the CAN Data. For CAN messages with ID 0x2B∗, the first byte of CAN Data being 0x40 or 0x80 indicates turning the vehicle external discharging system on or off.

6 Real-world Attacks Driven by IVIHunter

In this section, driven by IVIHunter's app analysis and CAN reverse engineering results, we reveal remote vehicle status monitoring and malicious vehicle control attacks targeting the BYD Qin vehicle.

6.1 Remote Vehicle Status Monitoring

We launch the monitoring attack through CAN traffic hijacking and translation. Since IVIHunter has accomplished CAN reverse engineering, the translation can be achieved by checking whether CAN messages for specific vehicle control actions appear in the traffic. Therefore, our focus lies in hijacking the CAN traffic.

6.1.1 Zero-permission Attacker Threat Model. In this attack, the adversary is limited to a normal capability, meaning they cannot physically access the vehicle but can install a zero-permission app on the head unit. This attack capability can be achieved by publishing malicious apps on third-party or official app stores (e.g., the BYD in-vehicle app store and Google Play, as detailed in Section 3.2) and luring victims to install them on the head unit. For instance, attackers can employ app squatting, which involves using app names, package names, or icons similar to those of popular apps to gain user trust and be installed on the head unit [13, 45]. Another potential approach is the app repackaging attack, where attackers inject malicious code into a legitimate app and repackage it for distribution [89, 93]. Optionally, to strengthen attack performance (see Section 6.1.3), the adversary operates a wireless access point to lure the victim head unit into connecting, such as by impersonating a public Wi-Fi network.

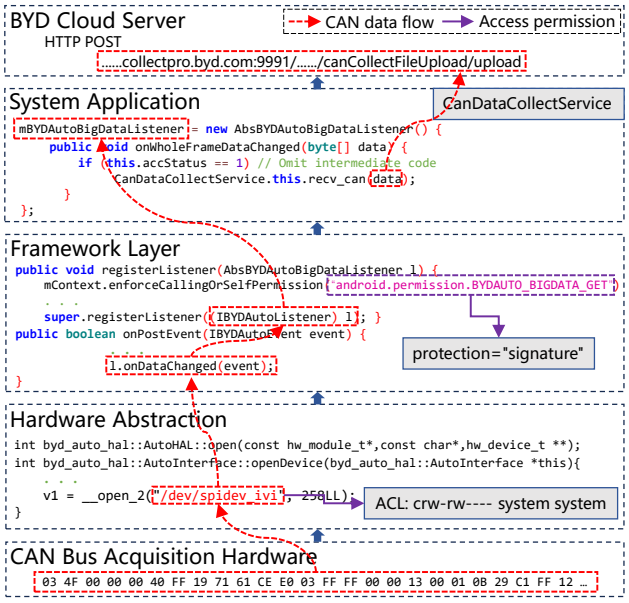


Figure 7: Workflow for collecting and uploading CAN traffic.

6.1.2 Standard Vehicle Status Monitoring. The In-vehicle App Analysis Module of IVIHunter is leveraged to scan potential CAN data collection apps and perform analysis on them. Concretely, we designate Android network request APIs as entry functions and conduct backward program slicing to identify the CAN traffic collection and upload process. As shown in Figure 7, DiLink receives in-vehicle CAN data through the */dev/spidev_ivi* device file. The process is monitored by the framework layer, which transmits the CAN data to system apps. In DiLink, the privileged system app *CanDataCollectService* continuously uploads CAN data to BYD's cloud server every minute, which is for purposes like anomaly detection.

However, our zero permission assumption prevents direct attacks on the workflow shown in Figure 7. Primarily, since we cannot physically access the CAN bus, direct sniffing is infeasible. Moreover, access to */dev/spidev_ivi* device is restricted by the access control list (ACL) to system users, and framework-layer APIs require `signature` level system permissions. Given these constraints, the *CanDataCollectService* system app is targeted as the attack surface.

We further examine the internal details of the program slicing function chain extracted by IVIHunter and identify that *CanDataCollectService* configures the URL for CAN traffic uploading upon receiving a broadcast with action label *com.byd.data_collection_notify* and retrieves the URL from *can_path* field, as illustrated in Figure 8.

```
BroadcastReceiver mReceiver = new BroadcastReceiver(){
    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent){
        // Omit intermediate code
        if ("com.byd.data_collection_notify".equals(action)) {
            String upload_url = intent.getStringExtra("can_path");
            CanDataCollectService.this.updateUploadUrl(upload_url);
        }
    }
};
```

Unprotected Broadcast

Update CAN Traffic Upload URL

Figure 8: CanDataCollectService app vulnerability code.

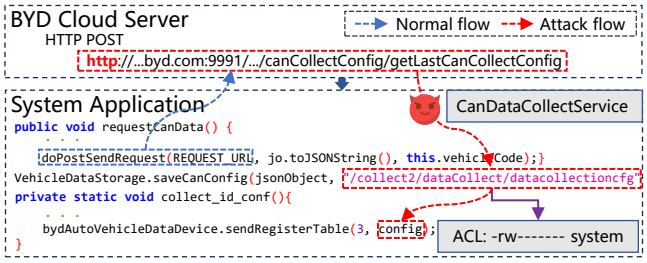
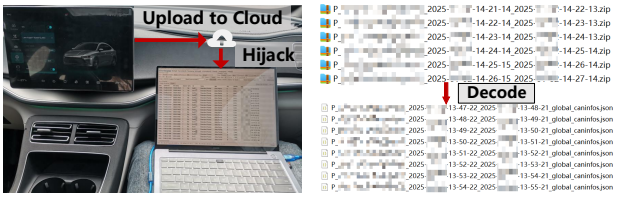


Figure 9: Workflow for configuring traffic collection rules.



(a) Real vehicle attack environment. (b) Hijacked CAN traffic results.

Figure 10: Real-vehicle experiment environment and results.

The vulnerability arises because the broadcast is not set as protected [34]. Therefore, any app can send such a broadcast, enabling an attacker to manipulate the upload URL and redirect CAN data to an attacker-controlled server, resulting in CAN traffic hijacking. We attribute the vulnerabilities to implementation flaws, where manufacturers attempt to enforce isolation through OS-level permission mechanisms, yet introduce critical weaknesses.

6.1.3 Strengthened Vehicle Status Monitoring. Based on above analysis, we already achieved remote monitoring. To strengthen the attack, we dive into program slicing results and reveal that *CanDataCollectService* only captures messages with specific CAN IDs according to a configuration file, which leads to discarding some CAN traffic required for eavesdropping on specific vehicle status.

To broaden monitoring scope, we exploit that *CanDataCollectService* adopts the HTTP protocol to request a configuration file during vehicle startup, as presented in Figure 9, which allows an attacker to masquerade as a public Wi-Fi to lure the victim head unit into connecting, and then launch the man-in-the-middle attack by impersonating the cloud server to deliver a spoofed response. Ultimately, leveraging IVIHunter's CAN reverse engineering results, the attacker can modify the configuration so that CAN messages associated with targeted vehicle control states are recorded in the hijacked CAN traffic, thereby achieving strengthened monitoring.

6.1.4 Real Vehicle Evaluation Result. We test attacks on a BYD Qin Plus 2021 vehicle shown in Figure 10. The result provided in Table 8 reveals that with a standard monitoring attack, 15 vehicle states are remotely monitored, along with related CAN messages, while

Table 8: CAN message indicating the change of vehicle status.

Vehicle Status	CAN Message Indicating Vehicle Status Change	
	CAN ID	CAN Data
Lane Assist System Mode	0x31∗	(048C)1 C0 ** ** ** ** ** **
Traffic Sign Recognition		** ** FF (15)F ** ** ** **
Intelligent High/Low Beam		(01)0 C0 ** ** ** ** ** **
Predictive Collision Warning	0x32∗	** ** 0(04) ** ** ** ** **
Automatic Emergency Braking		** ** ** ** (EF)4 ** ** **
Energy Recycle Mode	0x34∗	** ** 12 0(9D) ** ** ** **
Set State of Charge to 25%		** ** ** ** 19 ** ** **
Brake Pedal Adjustment	0x12∗	(9B)2 C0 ** ** ** ** ** **
Steering Assistance Mode	0x24∗	FC 3(13) ** ** ** ** ** **
Comfortable Parking Mode	0x0D∗	** ** 4(01) 80 ** ** ** **
Blind Spot Detection	0x41∗	E(01) 80 ** ** ** ** ** **
Charger Locker Mode	0x47∗	(65)0 ** ** ** ** ** ** (78)5
Ex-Mirror Auto Folding	0x40∗	** ** (12)8 20 ** ** ** **
Adjust Lamp Height	0x38∗	** ** 90 ** ** ** ** B7
Rear Defrost	0x43∗	** ** ** ** 8(89) 0(64) ** **
Air Conditioning Switch	0x40∗	9E 5(69) ** ** ** ** ** **
Air Conditioning Cooling Mode		E(57) 55 ** ** ** ** ** **
Automatic Cooling Mode		E(35) 5(95) ** ** ** ** ** **
Maximum Cooling Mode		** ** (73)1 (56)F ** ** ** **
Split Control Mode		** ** 13 (56)F ** ** ** **
Front Defrost		** (59)5 ** (56)F ** ** ** **
Air Circulation Mode		** (56)5 13 ** ** ** ** **
Ventilation switch		** ** 1(13) (56)F ** ** ** **
Set Driver/Passenger to 25°C		** ** ** ** 19 19 ** **
Wind Direction Mode		** ** 2(247) 6F ** ** ** **
Set Blowing Strength to 1		** ** (1)∗ ** ** ** ** **

the different values within brackets in the CAN data represent the varying states of the corresponding vehicle attributes. For example, the head unit collects CAN messages with CAN ID 0x32∗, we can check whether the fifth byte of the CAN Data is 0xE4 or 0xF4 to determine if the automated emergency braking is enabled. After incorporating 11 types of collected messages by adopting a strengthened attack, the number of monitored states increases to 26. For instance, we can add a rule to collect CAN messages with ID 0x41∗ to monitor the blind spot detection system. The vulnerabilities have been disclosed to BYD, confirmed, and assigned CVE IDs.

6.2 Malicious Injection During Assisted Driving

In this subsection, we introduce how IVIHunter facilitates the injection of malicious vehicle control commands. Specifically, we focus on the safety-critical ADAS.

6.2.1 Privileged Attacker Threat Model. We assume that privileged access has been obtained through vulnerabilities like permission escalation, as demonstrated by numerous prior attacks against different vehicle brands [26, 36, 40, 50, 55, 66, 76, 81]. For example, an attack vector is to exploit privilege escalation vulnerabilities originating from the native OS (e.g., Android and AAOS), which are potentially inherited by the head unit OS [33]. Our focus is not on how this assumption is achieved, but on the observation that privileged permissions do not directly imply vehicle control capability. Instead, attackers still need to identify feasible attack paths to inject malicious commands [26, 36, 50, 55, 62, 66, 76], which typically involves extensive manual efforts, such as reverse engineering OSs to uncover vehicle control mechanisms, analyzing hardware to understand in-vehicle network architectures, and parsing communication protocols to construct valid commands. Therefore, IVIHunter aims to eliminate these tasks to enable efficient attacks.

Table 9: Hidden API exploitation and resultant ADAS control.

ADAS Control Action	The Corresponding Hidden API			
	Class	API Name	Parameter	
			Enable	Disable
Intelligent Cruise Control	***.BYDAuto ADASDevice	setTJAState	1	0
Lane Keeping Assist		setLKSMode	3	0
Lane Departure Warning		setLDSWType	1	0
Traffic Sign Recognition		setSLAState	1	0
Predictive Collision Warning		setPCWState	1	0
Automatic Emergency Braking		setAEBState	1	0
Blind Spot Detection		setBSDState	1	0
Electronic Body Stabilization		setESPState	1	0
Reverse Parking Sensor	***.BYDAuto RadarDevice	setReverseRadar SwitchState	1	0
Smart Matrix Light	***.BYDAuto LightDevice	setAcLight AdbState	1	2

6.2.2 Hidden API Identification and Exploitation. In-vehicle apps execute vehicle control actions by invoking corresponding system APIs. However, due to the closed-source nature of head unit OSs, these APIs are hidden. Driven by IVIHunter, we examine the results of In-vehicle App Analysis Module to uncover these hidden APIs.

Concretely, resource identifiers of the extracted vehicle control components are examined to recognize ADAS-relevant objects through semantic analysis (e.g., matching the keyword "pilot"). Given that the listener functions bound to components have been parsed through backward program slicing and the app's control flow graph has also been constructed via the process in Section 4.2.2, IVIHunter performs forward control flow analysis from each listener function to system APIs, while filtering out irrelevant calls such as UI rendering. Finally, statement-level analysis is conducted on the identified functions to summarize the invocation chains of hidden vehicle control APIs and associated data transmission logic.

The following task is exploiting the results to launch malicious control, yet we cannot directly invoke the APIs due to the lack of the official SDK. To address this, we leverage Java's reflection mechanism [67], which enables dynamic access to hidden classes and functions via their signatures at runtime. Specifically, based on the extraction results, IVIHunter uses class signature reflection to obtain the ADAS class, subsequently invoking its instance construct and vehicle control APIs through function signature reflection, thereby achieving control over the ADAS.

6.2.3 Analysis Result. The result indicates that the hidden APIs for controlling 10 ADAS subsystems are extracted. Table 9 presents representative APIs and parameters for individual vehicle control actions (e.g., enabling or disabling specific functions), each corresponding to the final functions in the complete API invocation chain. For safety and ethical considerations, we flash the rooted OS [25] on a standalone BYD Qin head unit to conduct the attack validation. Specifically, we check whether CAN messages are triggered by the attack and align with results from CAN reverse engineering. Moreover, we replay CAN messages and observe vehicle states shown on the app UI, which should follow the malicious vehicle control actions. For example, when we maliciously disable the intelligent cruise control (ICC) system by invoking a hidden API, the corresponding CAN messages should be transmitted from the head unit, and the illuminated button in the `CarSettings` app shown in Figure 6 is supposed to transition to a deactivated (off) state. The evaluation confirms that all 10 malicious control attacks against ADAS are successful, each significantly undermining driving safety.

7 Discussion

Countermeasures. Our attack scheme consists of CAN reverse engineering and real-world attacks. For the former, app hardening and control flow obfuscation [10] are recommended by SAE J3061 [1] to thwart code analysis. However, such mechanisms increase app latency and memory overhead, which potentially weaken system stability. For the vehicle status monitoring attack, developers should designate the exploited broadcasts as protected to restrict their use to system apps. Additionally, the HTTPS protocol or the AUTOSAR standard [9] should be adopted to ensure secure communication. However, the introduced performance overhead may also undermine the real-time requirements of in-vehicle OSs. As to the malicious injection attack, the head unit's OS should always be updated with the latest security patches to prevent privilege escalation.

Future Work. IVIHunter is potentially applicable to other Android-based head units. For specific non-Android systems, a modular approach can be adopted. For example, although IVIHunter cannot perform app analysis and dynamic triggering, alternative methods like visual-based robotic arms based tools [91] can be adapted for the automated triggering task, with their outputs fed into IVIHunter's subsequent modules to complete analysis. A potential issue is that specific apps in the standalone head unit become non-interactive, which can be addressed by connecting the protocol box to the head unit to emulate a real vehicle. The protocol box is originally designed to support maintenance and in-vehicle app development, and can be publicly purchased from automotive repair shops.

8 Related Work

Attacks on Head Units. Mittal et al. [63] conduct an analysis of AAOS and reveal that its permission mechanism has overly coarse-grained defects. Jeong et al. [52] apply privacy leak detection tools for Android to detect privacy leaks. Pese et al. [68] analyze in-vehicle apps on Google Play and conclude that 80% of them have potential vulnerabilities. Mandal et al. [58] discover the denial of service risk in the Automotive Grade Linux system. Renganathan et al. [73] achieve data theft by exploiting Bluetooth vulnerabilities.

Attacks on CAN Protocol. CAN sniffing attacks [28, 57] exploit the broadcast feature and lack of encryption of CAN messages. Leveraging the lack of authentication, CAN injection attacks enable any access point to send malicious CAN messages [28, 57, 62]. Due to the CAN ID-based arbitration and limitations on bus bandwidth, DoS attacks are proven effective [56, 64]. Fuzz testing has also been applied to uncover unknown vulnerabilities [54, 57]. Given the substantial driving data in CAN traffic, fingerprinting attacks based on sniffing and analysis have been proposed [51].

Reverse Engineering of CAN Protocol. READ [59] identifies CAN messages by observing sharp drops in the bit-flipping rate within traffic, and infers semantics by analyzing the CAN Data variation trend with domain knowledge. LibreCAN [59] builds upon READ by enhancing boundary recognition and semantic identification algorithms. AutoCAN [39] leverages the correlations between signal values in CAN messages to delineate field boundaries and recovers semantics based on physical laws. CANHunter [87] analyzes companion diagnostic apps to extract hard-coded CAN messages. To remove manual operations, DP-Reverser [90] utilizes a robotic arm to interact with professional diagnostic devices.

9 Conclusion

In this paper, we reveal cross-domain attacks launched from the head unit of modern vehicles. To perform CAN reverse engineering, IVIHunter, an automated analysis tool, is designed. Evaluation conducted on four head unit OSs demonstrates that IVIHunter extracts 629 CAN messages corresponding to 230 vehicle control functions. Driven by IVIHunter, we exploit vulnerabilities to launch cross-domain remote monitoring and malicious control attacks. The vulnerabilities have been confirmed and assigned CVE IDs.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (No. 62325207, 62132013, 62302298).

References

[1] 2021. Cybersecurity Guidebook for Cyber-Physical Vehicle Systems. https://www.sae.org/standards/content/j3061_202112/.
[2] 2024. Road vehicles — Controller Area Network (CAN). https://www.iso.org/obp/ui/en/#iso:std:iso:11898:-1.
[3] American Honda Motor Co. 2025. Honda with Google built-in. https://automobiles.honda.com/google-built-in.
[4] Android Developers. 2025. AAOS VehiclePropertyIds Constants. https://developer.android.com/reference/android/car/VehiclePropertyIds#constants_1.
[5] Android Developers. 2025. Vehicle Microcontroller Unit Hardware architecture. https://source.android.com/docs/automotive/security/vehicle_system_isolation.
[6] Android Developers. 2025. Vehicle Microcontroller Unit Hardware architecture. https://source.android.com/docs/automotive/power#hardware.
[7] Android Developers. 2026. Test UI with UI Automator. https://developer.android.com/training/testing/ui-automator.html#ui-automator-viewer.
[8] AutoPi.io. 2025. CAN Sniffer: 5 Steps to Reverse Engineering the CAN Bus. https://www.autopi.io/blog/discover-hidden-functions-in-your-car-with-can-bus/.
[9] AUTOSAR Consortium. 2023. AUTOSAR Specifications, Release R23-11. https://www.autosar.org/standards/.
[10] Vivek Balachandran, Darell JJ Tan, Vrizlynn LL Thing, et al. 2016. Control flow obfuscation for android applications. *Computers & Security* 61 (2016), 72–93.
[11] Victor Bandur, Vera Pantelic, Timofey Tomashevskiy, and Mark Lawford. 2021. A safety architecture for centralized E/E architectures. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 67–70.
[12] Victor Bandur, Gehan Selim, Vera Pantelic, and Mark Lawford. 2021. Making the case for centralized automotive E/E architectures. *IEEE Transactions on Vehicular Technology* 70, 2 (2021), 1230–1245.
[13] Bleeping Computer. 2025. Malicious Android apps on Google Play downloaded 42 million times. https://www.bleepingcomputer.com/news/security/malicious-android-apps-on-google-play-downloaded-42-million-times/.
[14] BMW Group. 2023. BMW Group expands BMW Operating System 8, integrates Android Automotive OS. https://www.press.bmwgroup.com/global/article/detail/T0401875EN/bmw-group-expands-bmw-operating-system-8-integrates-android-automotive-os.
[15] Bosch Mobility. 2025. Central Gateway. https://www.bosch-mobility.com/en/solutions/vehicle-computer/central-gateway/.
[16] Bosch Mobility Solutions. 2024. Whitepaper: The Next Step in E/E Architectures. https://www.bosch-mobility.com/en/mobility-topics/ee-architecture/.
[17] BUICK. 2024. Stay connected with Google built-in compatibility. https://www.buick.com/explore/news/google-built-in.
[18] Ondrej Burkacky, Fabian Steiner, Martin Kellner, Johannes Deichmann, and Julia Werra. 2023. Getting ready for next-generation E/E architecture with zonal compute. *McKinsey & Company* 14 (2023).
[19] Alessio Buscemi, Ion Turcanu, German Castignani, Romain Crunelle, and Thomas Engel. 2021. CANMatch: A Fully Automated Tool for CAN Bus Reverse Engineering Based on Frame Matching. *IEEE Transactions on Vehicular Technology* 70 (2021). doi:10.1109/TVT.2021.3124550
[20] BYD. 2021. BYD App Store Management Guidelines. https://oip.byd.com/addons/cms/document/index.
[21] BYD. 2022. BYD ATTo OWNER'S MANUAL. https://www.byd.com/content/dam/byd-site/za/product/atto3/attopdf/BYDAtto3_Brochure.pdf.
[22] BYD. 2022. BYD Support. https://www.byd.com/en-ma/support/dilink.
[23] BYD. 2023. Why is BYD's e-Platform 3.0 so special? https://www.byd.com/eu/blog/Why-is-BYDs-e-Platform-3-0-so-special.
[24] BYD. 2024. BYD DiLink. https://www.byd.com/en-ma/support/dilink.
[25] BYDcar. 2023. BYD (Build Your Dream) Car Repair Manuals and Factory Images. https://github.com/BYDcar/BYDPackagesByChip2.
[26] Zhiqiang Cai, Aohui Wang, Wenkai Zhang, Michael Gruffke, and Hendrick Schweppe. 2019. 0-days & mitigations: Roadways to exploit and secure connected BMW cars. *Black Hat USA* 2019, 39 (2019), 6.
[27] CARIAD. 2021. *How new infotainment will shape the future customer experience.* https://cariad.technology/de/en/news/stories/infotainment-customer-experience.html
[28] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX security symposium (USENIX Security 11)*.
[29] LYNK & CO. 2025. Lynk & Co 06. https://lynkco-sa.com/en/lynk-co-06/.
[30] Geely Automobile International Corporation. 2025. OWNER MANUAL GEELY STARRAY EM-i. https://geelyprod-static.oss-ap-southeast-5.aliyuncs.com/owner-book/geely_starray_em_i_owner_manual.pdf.
[31] Volvo Car Corporation. 2024. New apps can be downloaded and installed when the car is connected to the Internet. https://www.volvocars.com/en-th/support/car/xc60/article/76b5f4d05cd696f5c0a8015147ff239a/.
[32] CSS Electronics. 2026. CAN Bus Sniffer - Reverse Engineer Vehicle Data [SavvyCAN/Wireshark]. https://www.csselectronics.com/pages/can-bus-sniffer-reverse-engineering.
[33] Cybersecurity Help. 2024. SB2024090433 - Privilege escalation in Android Automotive OS. https://www.cybersecurity-help.cz/vdb/SB2024090433.
[34] Android Developers. 2024. Insecure Broadcast Receiver. https://developer.android.com/privacy-and-security/risks/insecure-broadcast-receiver.
[35] Android Developers. 2025. Car ready mobile apps. https://developer.android.com/training/cars/car-ready-mobile-apps.
[36] Mikhail Evdokimov and Radu Motspan. 2025. Remote Exploitation of Nissan Leaf: Controlling Critical Body Elements from the Internet. In *Black Hat Asia*. Singapore.
[37] Ford Motor Company. 2025. Your Lincoln Digital Experience. https://www.lincoln.com/technology/lincoln-digital-experience/.
[38] Fortune Business Insights. 2026. Automotive Domain Controller Market Size, Share & Industry Analysis, By Domain (Powertrain, Body & Chassis, Infotainment, and ADAS), By Vehicle Type (Passenger Cars and Commercial Vehicles), By Propulsion (Electric and IC Engine), By Application (Active Safety, Body Control, User Experience, and Powertrain Management), and Regional Forecast, 2026-2034. https://www.fortunebusinessinsights.com/automotive-domain-controller-market-108408.
[39] Daniel Frassinelli, Sohyeon Park, and Stefan Nürnberger. 2020. I Know Where You Parked Last Summer : Automated Reverse Engineering and Privacy Analysis of Modern Cars. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1401–1415. doi:10.1109/SP40000.2020.00081
[40] Scott Gayou. 2018. Jailbreaking Subaru StarLink. https://github.com/sgayou/subaru-starlink-research.
[41] Geely. 2025. Geely Boundless Space. https://dh.geely.com/SS21.
[42] General Motors. 2025. 2025 Chevrolet Equinox EV Owner's Manual. https://contentdelivery.ext.gm.com/bypass/gma-content-api/resources/sites/GMA/content/staging/MANUALS/9000/MA9475/en_US/1.0/GTK_2025_Chevrolet_Equinox_EV_85657282_A.pdf.
[43] Google. 2024. Android Automotive. https://source.android.com/docs/automotive.
[44] Google. 2025. Guidelines for development. https://source.android.com/docs/automotive/guidelines.
[45] Yangyu Hu, Haoyu Wang, Ren He, Li Li, Gareth Tyson, Ignacio Castro, Yao Guo, Lei Wu, and Guoai Xu. 2020. Mobile app squatting. In *Proceedings of The Web Conference 2020*. 1727–1738.
[46] Hyundai Motor Company. 2024. Hyundai Motor Group and Google Collaborate on Software Capability for Future Mobility Innovation. https://www.hyundai.com/worldwide/en/newsroom/detail/0000000883.
[47] Hyundai Motor Company. 2025. Vehicle Settings — Infotainment System. https://ownersmanual.hyundai.com/docview/webhelp/Hyundai/928ab0a3-4871-4678-9c4c-e850886c0706/id7419db9bd35.html.
[48] Fortune Business Insights. 2026. Automotive Infotainment Market Size, Share & Industry Analysis, By Vehicle Type (Passenger Cars, Light Commercial Vehicles, and Heavy Commercial Vehicles), By Application (Navigation, Media, Communication, Payment Services, and Telematics), By Distribution Channel (OEM and Aftermarket), and Regional Forecast, 2026-2034. https://www.fortunebusinessinsights.com/automotive-infotainment-market-102676
[49] International Organization for Standardization. 2021. Road vehicles — Cybersecurity engineering. https://www.iso.org/standard/70918.html.
[50] Artem Ivachev and Danila Parnishchev. 2024. Over the Air: Compromise of Modern Volkswagen Group Vehicles. In *Black Hat Europe*. London, United Kingdom.
[51] Saja Falah Jabbar, Laith Ali Abdul-Rahaim, and Monir Ali Lilo. 2024. Driver Identification Via Hybrid Model and Few-Shot Technique Based CAN-BUS data. *International Journal of Transport Development and Integration* 8, 3 (2024).
[52] Seonghoon Jeong, Minsoo Ryu, Hyunjae Kang, and Huy Kang Kim. 2023. Infotainment System Matters: Understanding the Impact and Implications of In-Vehicle

Infotainment System Hacking with Automotive Grade Linux. In *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*. 201–212.
[53] Shiho Kim and Rakesh Shrestha. 2020. *In-Vehicle Communication and Cyber Security*. Springer Singapore, 67–96.
[54] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2010. Experimental security analysis of a modern automobile. In *2010 IEEE symposium on security and privacy*. IEEE, 447–462.
[55] Tencent Keen Security Lab. 2021. *Mercedes-Benz MBUX Security Research Report*. Technical Report. Tencent Keen Security Lab.
[56] Yousik Lee and Samuel Woo. 2022. CAN Signal Extinction-based DoS Attack on In-Vehicle Network. *Security and Communication Networks* 2022, 1 (2022), 9569703.
[57] Jiajia Liu, Shubin Zhang, Wen Sun, and Yongpeng Shi. 2017. In-vehicle network attacks and countermeasures: Challenges and future directions. *IEEE Network* 31, 5 (2017), 50–58.
[58] Amit Kr Mandal, Agostino Cortesi, Pietro Ferrara, Federica Panarotto, and Fausto Spoto. 2018. Vulnerability analysis of android auto infotainment apps. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*.
[59] Mirco Marchetti and Dario Stabili. 2019. READ: Reverse Engineering of Automotive Data Frames. *IEEE Transactions on Information Forensics and Security* 14, 4 (2019). doi:10.1109/TIFS.2018.2870826
[60] Michael W. 2017. How to hack a car — a quick crash-course. https://www.freecodecamp.org/news/hacking-cars-a-guide-tutorial-on-how-to-hack-a-car-5eafcfbbb7ec.
[61] Charlie Miller and Chris Valasek. 2014. A Survey of Remote Automotive Attack Surfaces. In *Black Hat USA*. Las Vegas, NV.
[62] Charlie Miller and Chris Valasek. 2015. Remote exploitation of an unaltered passenger vehicle. In *Black Hat USA*. Las Vegas, NV.
[63] Pooja Mittal, Ramakrishnan Naryanan, Tanya Agarwal, and Vivek Agrawal. 2022. *Investigation of Privacy Monitoring Systems in the Android Device-A Case Study of Automotive IVI System*. Technical Report. SAE Technical Paper.
[64] Pal-Stefan Murvay and Bogdan Groza. 2018. Security shortcomings and countermeasures for the SAE J1939 commercial vehicle bus protocol. *IEEE Transactions on Vehicular Technology* 67, 5 (2018), 4325–4339.
[65] Varun M. Navale, Kyle W. Williams, Athanassios Lagospiris, Michael Schaffert, and M. Schweiker. 2015. (R)evolution of E/E Architectures. *SAE International Journal of Passenger Cars - Electronic and Electrical Systems* 8 (2015), 282–288.
[66] Sen Nie, Ling Liu, and Yuefeng Du. 2017. Free-fall: Hacking tesla from wireless to can bus. *Briefing, Black Hat USA* 25, 1 (2017), 16.
[67] Oracle. 2024. Java Reflection API Reference. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html
[68] Mert Pese, Kang Shin, Josiah Bruner, and Amy Chu. 2020. Security analysis of android automotive. *SAE International Journal of Advances and Current Practices in Mobility* 2, 2020-01-1295 (2020), 2337–2346.
[69] Mert D. Pesé, Troy Stacer, C. Andrés Campos, Eric Newberry, Dongyao Chen, and Kang G. Shin. 2019. LibreCAN: Automated CAN Message Translator. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom). Association for Computing Machinery, New York, NY, USA, 2283–2300. doi:10.1145/3319535.3363190
[70] Polestar. 2024. Polestar 2 Download apps. https://www.polestar.com/us/manual/polestar-2/2024/article/d61b349459320b5fc0a8015105e15000/.
[71] Pulse Security. 2021. Practical CANBUS Reversing - Understanding the Ducati Monster. https://pulsesecurity.co.nz/articles/ducati-696-canbus.
[72] Renault. 2024. Google Play Store and apps. https://www.user-manual.renault.com/en/notice/16050/video/83337.
[73] Vishnu Renganathan, Ekim Yurtsever, Qadeer Ahmed, and Aylin Yener. 2022. Valet attack on privacy: a cybersecurity threat in automotive Bluetooth infotainment systems. *Cybersecurity* 5, 1 (2022), 30.
[74] ResearchInChina. 2025. Automotive Cockpit Domain Controller Research Report, 2025. http://www.researchinchina.com/Htmls/Report/2025/77081.html.
[75] Lynk & CO Automobile Sales. 2020. User Manual. https://chats.chinamobil.ru/files/lynk-co-russia/applicationpdf/LYNK06.pdf.
[76] Yuriy Serdyuk and Alexey Kondikov. 2022. Back-connect to the connected car. Search for vulnerabilities in the VW electric car.. In *Black Hat Europe*. London, United Kingdom.
[77] Craig Smith. 2016. *The car hacker's handbook: a guide for the penetration tester*. no starch press.
[78] Settaluri Lakshmi Sravanthi, Ankit Mishra, Debjyoti Mondal, Subhadarshi Panda, Rituraj Singh, and Pushpak Bhattacharyya. 2025. From Perception to Reasoning: Enhancing Vision-Language Models for Mobile UI Understanding. In *Findings of the Association for Computational Linguistics: ACL 2025*. 25250–25269.
[79] U.S. District Court for the Southern District of Florida. 2024. Class Action Complaint, Chicco v. General Motors LLC. Case No. 9:24-cv-80281-AMC. Filed Mar. 13, 2024.
[80] Nissan USA. 2025. NissanConnect Services with Google built-in. https://www.nissanusa.com/owners/connect/features-apps/google-services.html.
[81] VicOne. 2023. Two Tesla Hacks Triumph at Pwn2Own Vancouver 2023. https://vicone.com/blog/two-tesla-hacks-triumph-at-pwn2own-2023.
[82] N Vignesh, Mitesh Kumar, Balasubramanian Achuthan, Shivaji Badade, Prakash Shivakumar, and Ajeet Keshri. 2023. *Centralized E/E Architecture and Evolution*. Technical Report. SAE Technical Paper.
[83] VirusTotal. 2026. VirusTotal Online Malware Analysis Service. https://www.virustotal.com/.
[84] Volkswagen Newsroom. 2025. The new T-Roc: Interior and controls. https://www.volkswagen-newsroom.com/en/the-new-t-roc-world-premiere-19761/interior-and-controls-19765.
[85] Volvo Car Corporation. 2025. Explore XC40 features. https://www.volvocars.com/us/cars/xc40/features/.
[86] Volvo Car Corporation. 2026. Android emulator. https://developer.volvocars.com/in-car-apps/android-emulator-xc40/.
[87] Haohuang Wen, Qingchuan Zhao, Qi Alfred Chen, and Zhiqiang Lin. 2020. Automated Cross-Platform Reverse Engineering of CAN Bus Commands from Mobile Apps. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS'20)*. San Diego, CA.
[88] Wikipedia. 2025. Android Automotive. https://en.wikipedia.org/wiki/Android_Automotive.
[89] Yiming Wu, Zhuowu Liu, Yanjie Lin, Binbin Zhao, Chunyi Zhou, Tiejun Wu, and Zhen Hong. 2026. Android App Repackaging Detection: A Comprehensive Survey. *Cyber Security and Applications* (2026), 100124.
[90] Le Yu, Yangyang Liu, Pengfei Jing, Xiapu Luo, Lei Xue, Kaifa Zhao, Yajin Zhou, Ting Wang, Guofei Gu, Sen Nie, and Shi Wu. 2022. Towards Automatically Reverse Engineering Vehicle Diagnostic Protocols. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA.
[91] Shengcheng Yu, Chunrong Fang, Mingzhe Du, Yuchen Ling, Zhenyu Chen, and Zhendong Su. 2024. Practical non-intrusive GUI exploration testing with visual-based robotic arms. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[92] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. Appagent: Multimodal agents as smartphone users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–20.
[93] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *2012 IEEE Symposium on Security and Privacy*. 95–109. doi:10.1109/SP.2012.16
[94] ZHUHAI XINGJI MEIZU INFORMATION TECHNOLOGY CO.,LTD. 2023. Flyme Auto. https://www.flymeauto.com/.

A Ethical Considerations

This work received approval from the IRB of the authors' institution. All experiments were conducted on equipment purchased by us, with original vehicles corresponding to head units already decommissioned. Moreover, before our purchase, all privacy information in head units had been erased. Any attacks performed on real vehicles were fully reverted to pre-experiment states. We have reported CAN analysis results to related manufacturers and masked CAN data in the paper. Additionally, upon discovering vulnerabilities, we disclosed them to the manufacturer. The vulnerabilities have since been resolved in updated versions before the paper submission. We have also reported them to public vulnerability databases.

B Open Science

We release the source code of IVIHunter utilized during the following process as artifacts. Firstly, we provide the vehicle control app analysis code discussed in Section 4.2, which extracts detailed information about the vehicle control components (such as resource identifiers, variable names, etc.). Moreover, the code for the automated vehicle control functions triggering task presented in Section 4.3 is included. The artifacts are available at https://github.com/rainymoods/IVIHunter.