CANvas: Open-Source CAN-Bus Simulation and Modular Cybersecurity Experimentation

Bridging Environmental Dynamics, Driver Behavior and Attack Simulation for Comprehensive Automotive Security

Kacem Hakim

Research and Development project owner:
Kacem Hakim

Master thesis submitted under the supervision of
Professors Pelsser Cristel and Sadre Ramin

in order to be awarded the Degree of
Master in Cybersecurity
Cryptalalysis and Forensics

Academic year
2024-2025

We hereby confirm that this thesis was written independently without the use of any sources beyond those cited, and all passages and ideas taken from other sources are cited accordingly.

We give permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use.

02/06/2025

Title: CANvas: Open-Source CAN-Bus Simulation and Modular Cybersecurity Experimentation

Author: Kacem Hakim

Master in Cybersecurity – Cryptanalysis and Forensics
Academic year: 2024-2025

Abstract

The increasing software complexity of modern vehicles, particularly the use of networked Electronic Control Units (ECUs) over the Controller Area Network (CAN) bus, has expanded the automotive cybersecurity attack surface. Despite this, researchers often lack accessible and reproducible environments that accurately reflect ECU interactions and vehicle behavior under cyber-physical attack scenarios.

This thesis introduces CANvas, an open-source simulation platform designed to fill this gap. CANvas models eight core ECUs—covering powertrain, stability, steering, and body control—each implemented as an independent thread with realistic CAN timing and communication. It features a modular attack framework for evaluating scenarios such as spoofed sensor inputs and unauthorized actuator commands.

A graphical dashboard provides live visualization of internal vehicle states, and structured export capabilities enable reproducibility and offline analysis. While limited in physical dynamics and ECU diversity, CANvas serves as a practical foundation for controlled experimentation in automotive cybersecurity research.

Preface

My first encounter with automotive cybersecurity came through a blog post entitled *"Hacking Kia: Remotely Controlling Cars With Just a License Plate"* [1]. While that attack did not target the in-vehicle CAN bus, the idea that a few lines of code could unlock—or even start—a modern vehicle immediately captured my attention.

Shortly thereafter, I noticed that our department was offering a master's thesis on automotive security. The timing felt ideal, and I eagerly seized the opportunity.

What began as casual curiosity quickly evolved into a deeper research question:

> *Can we offer security engineers an accessible yet realistic sandbox to study CAN-level attacks and their countermeasures?*

This thesis is my attempt to explore that question. Chapter 2 reviews the landscape of automotive threats and tooling gaps; Chapters 3 and 4 describe the architecture and implementation of **CANvas**, the open-source platform developed to simulate both ECUs and attacker behavior; Chapter 5 demonstrates how CANvas enables reproducible experiments, from rain-triggered phantom braking to spoofed driver inputs; and finally, Chapter 6 reflects on limitations and proposes directions for future work.

I hope this work helps lower the entry barrier for students, researchers, and enthusiasts interested in exploring the fast-evolving field of automotive cybersecurity—just as that Kia article lowered it for me.

Acknowledgements

I would really like to thank my supervisors, **Prof. Dr. Pelsser Cristel** and **Prof. Dr. Sadre Ramin**, for their clear guidance and constructive feedback during this work. At times, I found myself caught up in the technical depth of the work—eager to implement every possible feature and perfect every aspect of the tool. This ambition, while motivating, occasionally led to a loss of focus. I am grateful to them for helping me regain direction when I needed it most and for reminding me to prioritize impact and coherence over excessive complexity.

I am also grateful to the authors of the research papers cited throughout this thesis; their publications provided the technical foundation that made CANvas possible.

Finally, thanks to my friends and family for their steady support while I completed this project.

Use of Artificial Intelligence in the Thesis

Artificial Intelligence (AI) tools were selectively used throughout this thesis to support code review, technical writing, and structural consistency. Specifically, AI assisted in refactoring and verifying code modules, reformulating complex technical explanations, and ensuring coherence across chapters. At every stage, AI-generated content was manually reviewed, validated, and adapted to align with the objectives of the project. All sources, whether academic, open-source, or web-based, have been properly cited. This thesis upholds academic integrity, and no content has been included without critical oversight or original contribution.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ABS	Anti-lock Braking System
ACC	Adaptive Cruise Control
ACK	Acknowledgment
ADAS	Advanced Driver Assistance System
AEB	Autonomous Emergency Braking
API	Application Programming Interface
BCM	Body Control Module
BTM	Behavioral Threat Model
CAN	Controller Area Network
CAN FD	Controller Area Network Flexible Data-Rate
CAN XL	Controller Area Network eXtended Length
CiA	CAN in Automation
CRC	Cyclic Redundancy Check
CSV	Comma-Separated Values
DBC	Database CAN (vector message specification)
DLC	Data Length Code
DoS	Denial of Service
DSRC	Dedicated Short-Range Communications
ECU	Electronic Control Unit
EOF	End of Frame
EON	Experimental Openpilot Node
EPS	Electric Power Steering
ESP	Electronic Stability Program
FD	Flexible Data-rate
GUI	Graphical User Interface
HiL	Hardware-in-the-Loop
HSM	Hardware Security Module
HUD	Head-Up Display
IDS	Intrusion Detection System
IMU	Inertial Measurement Unit

ISO	International Organization for Standardization
ISO-TP	ISO 15765-2 Transport Protocol
JSON	JavaScript Object Notation
LIN	Local Interconnect Network
LKAS	Lane Keeping Assist System
LSB	Least Significant Bit
MAC	Message Authentication Code
MACsec	Media Access Control Security
MSB	Most Significant Bit
NVRAM	Non-Volatile Random-Access Memory
OBD-II	On-Board Diagnostics II
OEM	Original Equipment Manufacturer
OTA	Over-The-Air
PCM	Powertrain Control Module
PHY	Physical Layer
REC	Receive Error Counter
RTR	Remote Transmission Request
SCM	Steering Column Module
SecOC	Secure Onboard Communication
SHE	Secure Hardware Extensions
SiL	Software-in-the-Loop
SOF	Start of Frame
TEC	Transmit Error Counter
TLS	Transport Layer Security
TPMS	Tire Pressure Monitoring System
UDS	Unified Diagnostic Services
vCAN	Virtual Controller Area Network
VSA	Vehicle Stability Assist
YAML	YAML Ain't Markup Language

Chapter 1
Introduction

The automotive industry's shift towards interconnected, software-defined vehicles has significantly increased cybersecurity risks. Modern vehicles rely heavily on Electronic Control Units (ECUs), which communicate primarily via the Controller Area Network (CAN)—a legacy protocol originally designed for reliability but lacking critical security features such as authentication and encryption. As vehicles integrate more automated functionalities and connectivity features, vulnerabilities inherent in CAN bus communication are increasingly exposed, creating real-world security threats. Despite emerging standards and regulatory frameworks requiring rigorous cybersecurity assessments (ISO/SAE 21434, UNECE R155), researchers and practitioners still face significant barriers, primarily due to the limited availability of low-barrier testing environments. This thesis addresses this challenge by developing **CANvas**, an open-source simulation platform designed to enable accessible experimentation of CAN-based cyber-physical attacks, aiming to facilitate both offensive exploration and defensive strategies in automotive cybersecurity.

1.1 Motivations

1.1.1 Context

The Controller Area Network was engineered for millisecond-level determinism: its priority-based arbitration and tiny frame overhead make it perfect for power-train and chassis control. Yet the same design minimalism omits *authentication, confidentiality, and freshness*—that is, the ability to verify who sent a message, keep its contents secret, and confirm the message's temporal validity (i.e., it wasn't replayed from earlier traffic). Even as next-generation vehicles adopt Ethernet backbones and zonal controllers [3], legacy CAN segments continue to gate brake, steering, and body functions—keeping them squarely in an attacker's cross-hairs.

Original Equipment Manufacturers (OEMs) now rank cybersecurity alongside functional safety as one of their most pressing mid-term challenges [5]. As a result, both red-team researchers tasked with probing these vulnerabilities and defenders responsible for mitigating them—require testbeds that are not only realistic but also affordable and accessible. As the electronic content of vehicles grows—from just 1% of total car cost in the 1970s to a projected 50% by 2030 [6]—the attack surface expands proportionally. This digital proliferation underlines the urgency of securing embedded automotive systems and justifies the need for experimental platforms like CANvas.

1.1.2 Problem Statement

Despite more than a decade of "car-hacking" research, the community still lacks a *single, open-source workspace* that lets engineers move seamlessly from raw–byte CAN injections to system-level, physics-aware analysis. Existing tools fall into three non-overlapping families:

1. **Packet generators** (e.g. fuzzers, replay scripts) deliver full control over CAN frames

but run in an environmental vacuum—dashboards remain static and no physical consequences are observable.

2. **Diagnostic harnesses** exercise Unified Diagnostic Services (UDS) or OEM test modes, yet ignore how those requests translate into vehicle motion or driver feedback.

3. **Lightweight visual simulators** render virtual speedometers or tell-tales, but embed only hard-coded message schedules and provide no interface for custom attack logic.

Each category solves a piece of the puzzle; none couples *protocol-level freedom* with *contextual realism*. The fragmentation leaves two principal stakeholder groups underserved:

- **Security researchers and red-teamers** who must trace how a single spoofed frame can cascade— for instance, causing phantom braking on a wet downhill.

- **Defensive engineers and IDS designers** who need labelled, repeatable data that reflects real driving situations, not laboratory abstractions.

Consequently, an opportunity exists for a reproducible, extensible platform that blends low-level CAN manipulation with high-fidelity vehicle behaviour.

1.2 Project Statement and Contributions

Following the problem statement, this thesis introduces **CANvas**[1]—a modular, attacker-oriented simulation and testing platform that embeds cyber-physical context into CAN security experimentation. CANvas is *not* an intrusion-detection system; rather, it is a sandbox in which offensive researchers and defensive engineers can observe how crafted bus traffic interacts with realistic vehicle behaviour.

Key contributions:

- *Unified dashboard for attack modules.* Building on, and extending, the open-source *canToT* PoC collection [7], the GUI auto-discovers modules for spoofing, fuzzing, DoS, diagnostic abuse, and more. A thread-safe launcher provides start/pause/stop lifecycle control.

- *Simulation Engine: AutoSim.* Derived from the educational *ICSim* [8] project but re-architected, AutoSim injects road gradient, weather, time-of-day, altitude, and traffic density into simulation runs, yielding deterministic yet richly variable scenarios on virtual CAN interfaces.

- *DBC-driven vehicle mapping.* CANvas integrates a parser that ingests industry-standard DBC files—structured descriptions of CAN messages and signals—and normalises signal parameters such as scaling factors, offsets, and byte layouts. This process produces structured JSON-based ECU maps that are consumed by our simulation engine, ensuring consistent message interpretation across the toolchain. To enable practical experimentation with real-world data, we leverage the open-source **OpenDBC** repository maintained by comma.ai [9], which provides an extensive and regularly updated collection of reverse-engineered DBC files for dozens of vehicle models. This integration simplifies porting CANvas to simulate a variety of real-world vehicles, including those from Toyota, Honda, Hyundai, and Volkswagen.

[1] Pronounced *canvas*.

- *Hybrid session workflow.* CANvas supports both fresh simulation setups and the ability to reload and modify previously saved environments. This is made possible by the structured JSON output generated from DBC-based vehicle mappings, which encapsulate signal definitions, ECU roles, and environmental parameters. This workflow not only enhances reproducibility for security experiments, but it can also enable iterative training and evaluation of intrusion detection systems all under consistent and controllable conditions.

- *Modular and extensible design.* CANvas is built with flexibility in mind. The framework also allows for seamless of additional attack payloads, environmental variables, and future extensions for telemetry or performance monitoring. Its modular plugin system means future add-ons can be incorporated with minimal changes to the core codebase, ensuring CANvas can grow alongside emerging needs.

By bridging message-level control with environment-level realism, CANvas fills a gap between abstract fuzzers and expensive testbeds.

1.3 Roadmap

The remainder of this thesis is organised as follows:

- **Chapter 2 — Background and Landscape of Automotive CAN Security**
  The chapter defines CAN terminology, catalogues documented attack vectors, reviews existing counter-measures, and benchmarks a dozen open-source and commercial toolchains. here we will dive deeper about existing toosl and we will talk about some surveys in-vehicle networks and automotive-security research.

- **Chapter 3 — Project Objectives and Requirements**
  Formalises the problem statement and derives functional as well as non-functional requirements. It then outlines CANvas's top-level architecture.

- **Chapter 4 — Implementation**
  Describes the concrete realisation of each subsystem: the robust DBC parser and JSON ECU maps, the enhanced physics engine and environment scripting, and the re-implemented ECU micro-architectures.

- **Chapter 5 — Attack Surface in CANvas**
  Introduces the tool's attack library and explains the real-time monitoring dashboard.

- **Chapter 6 — Conclusion, Limitations, and Future Work**
  Summarises the thesis contributions, discusses the current limitations and the next steps in our project.

Chapter Summary

In summary, this introductory chapter has framed the cybersecurity challenge and presented **CANvas** as an open, context-aware response. It has also mapped the structure of the thesis, giving the reader a clear guide to the progression from background review (Chapter 2) through requirements and design (Chapter 3), implementation details (Chapter 4),

applied attack scenarios (Chapter 5), and, finally, critical reflection and future outlook (Chapter 6). With these foundations in place, we now turn to **Chapter 2** to situate our work within the broader literature and tool landscape of automotive CAN security.

Chapter 2

Background and Landscape of Automotive CAN Security

Automotive cybersecurity revolves significantly around the Controller Area Network (CAN) protocol, which connects various Electronic Control Units (ECUs) within modern vehicles. Understanding CAN's vulnerabilities and the limitations of existing tools forms the basis for our thesis. To frame our project clearly, this chapter addresses several guiding questions:

a) **CAN protocol fundamentals** – How is CAN structured, and what essential security features does it lack?

b) **Known vulnerabilities and attacks** – Which types of cyber-attacks targeting the CAN protocol have been demonstrated, both academically and practically?

c) **Existing protective measures** – What defensive mechanisms currently exist for protecting CAN-based automotive systems?

d) **Current simulation tools and limitations** – Which tools are available to simulate and study CAN security, and what are their significant shortcomings?

By addressing these questions, we clearly outline the context and justify the need for a new, accessible platform like **CANvas**, capable of providing realistic and reproducible cyber-physical experimentation.

2.1 Notations and Key Concepts

2.1.1 From Industrial Prototype to Automotive Backbone

The Controller Area Network (CAN) began as a wiring-reduction idea at Bosch in the mid-1980s and has since grown into the default backbone for virtually every passenger vehicle. Table 2.1 highlights the milestones that chart this evolution.

Table 2.1: Timeline of key milestones in the evolution and regulation of the CAN family

Year	Milestone	Description	Ref.
1986	Initial concept	Bosch presents a lightweight two-wire, multi-master bus at the SAE Congress, replacing point-to-point ECU wiring.	[10]
1991	CAN 2.0 spec	Release of CAN 2.0 A (11-bit IDs) and 2.0 B (29-bit IDs), enabling both simple and gateway-rich networks.	[11]
1993	ISO 11898:1993	CAN becomes an ISO standard, covering both data-link and high-speed physical layers in one document.	[12]
1996	OBD-II mandate	U.S. law mandates the J1962 diagnostic connector on all light vehicles; multiple legacy protocols allowed.	[13]
2001/08	CAN-based OBD	EU EOBD gasoline-vehicle mandate effective MY 2001; U.S. requires CAN (ISO 15765-4) from MY 2008.	[14]
2003	ISO modular split	Series reorganised into ISO 11898-1 (data-link layer) and ISO 11898-2 (high-speed PHY) for greater modularity.	[15, 16]
2006	ISO 11898-3	Low-speed/fault-tolerant physical layer published as ISO 11898-3.	[17]
2012	CAN FD	Bosch introduces CAN FD, supporting payloads up to 64 bytes and optional data-phase speeds up to 5 Mbit/s.	[18]
2015	ISO 11898-1:2015	CAN FD frame format ratified in ISO 11898-1, preserving classical arbitration with flexible data-rate option.	[19]
2016	ISO 11898-2:2016	High-speed physical layer updated for CAN FD (improved symmetry up to 5 Mbit/s, optional selective wake-up).	[20]
2018	CAN XL in Motion	CiA special-interest group begins drafting CAN XL, featuring payloads from 1 to 2048 bytes and data-phase speeds starting at 10 Mbit/s, scalable up to 20 Mbit/s.	[21]
2024	ISO 11898-1:2024	CAN XL added to the ISO family, completing the third-generation CAN data-link standard.	[21]

Take-away. CAN has advanced through three major waves—classic 2.0, high-bandwidth FD, and the forthcoming XL—yet its original arbitration philosophy remains intact. Combined with the 1996 OBD-II mandate, this makes CAN both ubiquitous and perpetually interesting as an attack surface.

2.1.2 Functional layers inside a typical CAN node

CAN could scale so quickly and this is really demonstrated by theFig.2.1: the protocol cleanly separates responsibilities. Each composant has a seperate mission. The microcontroller deals only with application data. There is also a dedicated controller that handles framing and arbitration and the transceiver worries about voltage levels and EMC.

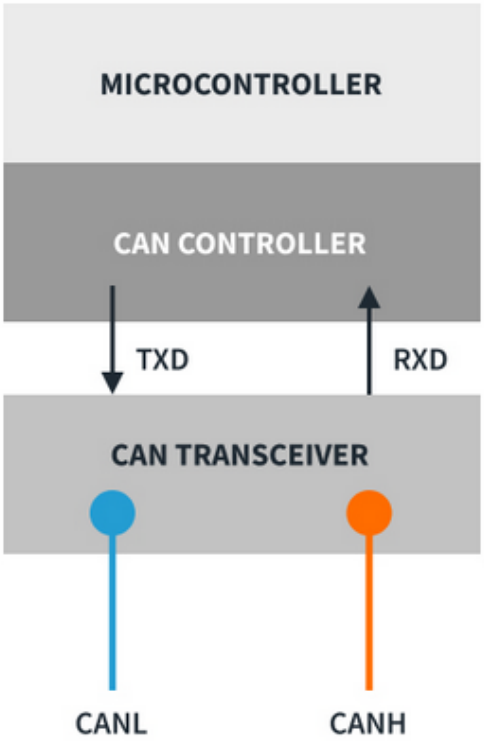

Figure 2.1: Layered composition of a CAN node [2]

2.1.3 Anatomy of a CAN Data Frame

Every classical CAN message follows the fixed bit sequence in Fig. 2.2, giving the bus deterministic timing and robust error detection [12]. Table 2.2 summarises the function of each field (dominant = 0, recessive = 1).

SOF	Message Identifier	RTR	IDE	r0	DLC	Data	CRC	ACK	EOF	IFS
1-bit Dominant	11-bit or 28-bit (Arbitration Field)	1-bit	1-bit	1-bit	4-bit	0 to 8 Bytes	15-bit Checksum 1 - bit Delimeter	1 - bit Acknowledgement 1 - bit Delimeter	7-bit Recessive	-

Figure 2.2: Classical CAN data-frame layout [3]

Table 2.2: Field-by-field breakdown of a classical CAN data frame (0–8-byte payload)

Field	**Purpose**
SOF	Dominant start bit that synchronises all nodes.
Arbitration	11- or 29-bit identifier + RTR; decides bus access—lowest ID wins.
Control	IDE flag and DLC nibble; announces payload length.
Data	0–8 B application payload (0–64 B in CAN FD).
CRC	15-bit checksum + delimiter for bit-level integrity.
ACK	Slot + delimiter; dominant slot = “frame received”.
EOF	Seven recessive bits spacing messages.

Looking Ahead

The next section builds on this foundation by mapping CAN's structural weaknesses to known attack techniques.

2.2 Automotive Threat Landscape and Known Vulnerabilities

Modern vehicles, heavily reliant on embedded electronics and connectivity, have become increasingly vulnerable to cyber threats. Like we mentioned in the earlier sections, while originally designed for robustness and fault tolerance, automotive communication systems—especially the Controller Area Network (CAN)—lack inherent security mechanisms such as authentication or encryption.

2.2.1 The Expanding Automotive Attack Surface

Modern connected vehicles feature numerous digital interfaces—both physical and wireless—each presenting potential entry points for attackers. Figure 2.3 illustrates the diversity and complexity of this landscape.

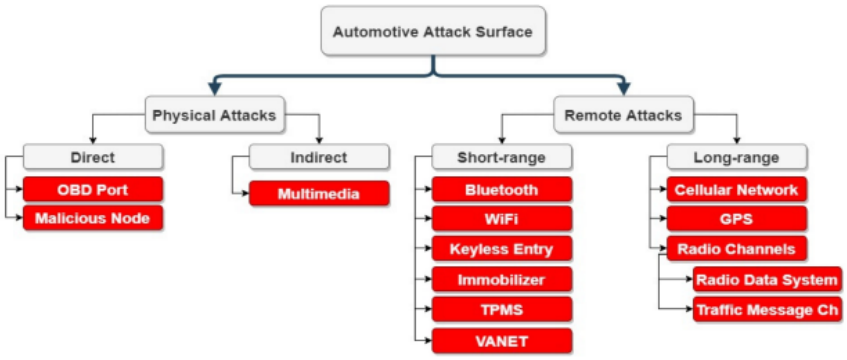

Figure 2.3: Comprehensive automotive attack surface, highlighting physical and remote vulnerabilities [3].

The automotive attack surface broadly divides into two categories:

Physical attacks

- **Direct attacks** involve explicit physical access to the vehicle's network, commonly through interfaces like the OBD-II diagnostic port. Researchers have demonstrated severe consequences, such as controlling critical vehicle functions (e.g., braking systems and engine control) by exploiting direct bus access through the OBD-II port [22].
- **Indirect physical attacks** occur through compromised external media, such as USB devices or multimedia systems. Such attacks involve implanting malicious firmware into infotainment units to gain internal network access [23] [24].

Remote attacks These attacks exploit wireless interfaces and remote services. Vehicles today have Bluetooth, cellular, Wi-Fi, and telematics systems integrated through gateway ECUs. Significant attacks include:

- **Cellular and telematics exploits**, notably the famous Jeep Cherokee incident (2015), where attackers remotely controlled steering, braking, and engine functions through vulnerabilities in the Uconnect infotainment system [25].
- **Over-The-Air (OTA) vulnerabilities**, demonstrated through ransomware attacks injected via OTA software updates, pose a severe risk due to their ability to scale widely without physical access [26].
- **Wireless-to-CAN exploits**, as shown in the cited Tesla Model S hack, involve chaining multiple vulnerabilities from wireless interfaces down to the CAN bus, effectively granting remote control of critical vehicle functions [27].

2.2.2 CAN Bus Vulnerabilities and Exploitation Techniques

The CAN protocol inherently prioritizes integrity and availability, but lacks security mechanisms such as encryption and authentication. This gap leaves CAN vulnerable primarily to two classes of attack:

- **Frame attacks**: These involve injecting well-formed CAN messages that are indistinguishable from legitimate traffic, enabling spoofing and replay attacks.
- **Protocol attacks**: More insidious and harder to detect, these involve directly manipulating CAN physical lines, sending carefully timed pulses to exploit subtle timing and electrical properties (e.g., Janus and Freeze-Doom Loop attacks demonstrated by Canis Labs through the CANHack toolkit [28]).

2.2.3 CAN Error Handling and Bus-Off Exploitation

The reliability of the Controller Area Network (CAN) protocol stems from its built-in error detection and confinement mechanisms. Each CAN node maintains two internal counters: the **Transmit Error Counter (TEC)** and the **Receive Error Counter (REC)**. These counters increment whenever a transmission or reception fault occurs and determine the node's error state, as shown in Fig. 2.4.

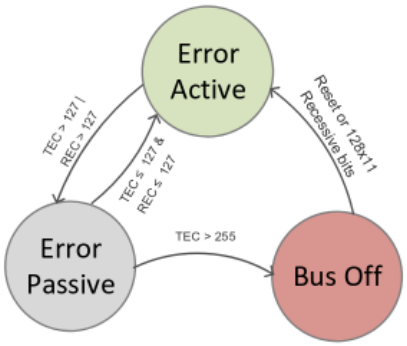

Figure 2.4: CAN error confinement mechanism (ECM) state transitions [3]

Error States Overview:

- **Error-Active:** Default state when both TEC and REC are below 128. The node participates normally in bus traffic and sends *dominant* error flags.

- **Error-Passive:** Triggered if either counter reaches or exceeds 128. The node must now send *recessive* error flags and waits before retrying transmissions, reducing its arbitration priority.

- **Bus-Off:** If the TEC exceeds 255, the node is forcibly disconnected from the bus. It cannot transmit or acknowledge any message until it either detects 128 occurrences of bus idle (recessive) bits or is reset via software.

Security Implications: The ECM mechanism, originally designed to isolate malfunctioning nodes, can be maliciously exploited. A determined attacker—especially one with physical access—can inject malformed frames or timing violations to increment a target ECU's TEC beyond 255. Once the ECU enters the Bus-Off state, it is effectively silenced, allowing the attacker to inject spoofed frames without interference. This strategy has been demonstrated in real-world CAN injection attacks, particularly in vehicles lacking runtime authentication or intrusion detection [29].

These considerations are essential for any simulation tool aiming to reproduce realistic CAN-level behaviour, particularly for research platforms that seek to combine signal-correct payloads, production-grade timing, and coherent vehicle states.

2.2.4 Catalog of Documented Automotive Attacks

Table 2.3 summarizes several well-documented attacks (some of them were cited above), emphasizing entry paths, compromised systems, and their outcomes.

Table 2.3: Representative documented automotive cyber attacks (2010–2024)

Name / Year	**Category**	**Attack Path**	**Compromised ECU**	**Impact**	**Reference**
OBD-II Exploit (2010)	Physical	Direct OBD-II	Power-train CAN	Disabled braking system at highway speeds via fuzzing OBD-II commands.	[22]
Malicious Node (2017)	Physical	Direct wiring	CAN bus node	DoS via overwriting recessive bits.	[24]
Jeep Uconnect (2015)	Remote	Cellular telematics	Infotainment Gateway	Remote control of critical safety systems (engine, brakes, steering).	[25]
Tesla Wireless-to-CAN (2017)	Remote	Wi-Fi	Gateway ECU	Full CAN access via chained wireless vulnerabilities.	[27]
Toyota Bus-Off (2022)	Physical	Direct CAN wiring	Headlamp module (body CAN)	Forced ECUs into Bus-Off, achieving isolated bus control.	[29]
OTA Ransomware (2017)	Remote	OTA update	ECU firmware updater	ECU firmware encrypted, ransom demanded for vehicle recovery.	[26]
Kia Web Vulnerability (2024)	Remote	Cloud API	Backend servers	Silent remote control via VIN; unauthorized user creation.	[1]

2.2.5 Countermeasures and Mitigation Strategies

Modern vehicle network security has evolved well beyond the notion of simply 'adding cryptography. Ongoing research in automotive cybersecurity highlight three persistent challenges that significantly influence the design of any CAN-based defense strategy:

a) **Physical access is still the top risk.**
Thieves now plug low-cost spoofers into exposed wiring (for example, a head-lamp harness) and inject CAN frames. In the 2023 RAV4 headlight attack (CVE-2023-29389) a $100 LIN-to-CAN bridge disabled the immobiliser and started the car within seconds [29]. For OEMs, every such weakness pushes theft insurance premiums higher until some models become **uninsurable** and even **unsellable**.

b) **SecOC cannot solve everything.**
AUTOSAR Secure On-board Communication (SecOC) adds message authentication, but it neither hides the payload nor fully prevents replay. Sequence counters stop reuse only while the ECU remains powered; a quick battery dip or soft reset clears the counter, letting an attacker resend old frames. Storing counters in NVRAM after each message is slow, costly, and shortens memory life—a concern practitioners raise often.

c) **Full encryption fits Ethernet, not classic CAN.**
The CiA IG-08 group is drafting *Ethernet-over-CAN FD* tunnels so MACsec can run at up to 10 Mbit/s. The draft is still pre-spec, and latency and overhead figures are only projections. Until supporting hardware arrives, stop-gap schemes such as CryptoCAN add secrecy but inherit SecOC's replay weakness.

The most robust designs therefore stack *five* complementary defences; each is outlined below:

a) **Cryptographic hardening** Lightweight Hardware Security Module (HSM)s sign a small set of critical CAN IDs, catching unauthorised tampering without slowing the whole bus [30].

b) **Secure transport layers**
SecOC already gives integrity; MACsec over CAN-FD tunnels will add secrecy and anti-replay once IG-08 silicon becomes available [31].

c) **Hardware-enforced integrity**
Analogue filters such as *CAN-HG* block bit-level protocol tricks (for example, Janus) before software processes the frame [32].

d) **Network segmentation with secure gateways**
Separating infotainment, Tire Pressure Monitoring System (TPMS), and power-train buses confines an intruder's reach—unless the gateway firmware itself is breached [33].

e) **Intrusion-detection systems (IDS)**
Anomaly-based IDS monitor rate, ID, and physical-layer patterns without changing CAN traffic, spotting zero-days at the cost of careful tuning [34].

Table 2.4: Layered countermeasures for securing CAN-based in-vehicle networks. Each layer targets a different threat; full protection calls for a combined approach.

Security Layer	Primary Protection	Strengths	Limitations
Frame-level Authentication (SecOC, CryptoCAN†)	Message integrity, sender proof	Software retrofit; $\sim$40 μs per frame; good for unlock/ignition commands	Adds 8–16 B overhead; keys required; replay possible after ECU reset
MACsec over CAN-FD Tunnel (CiA IG-08, draft)	Confidentiality, anti-replay, strong integrity	Uses familiar Ethernet stack; standardised; strong replay defence	Needs new gateway/PHY; limited to backbones $\leq$ 10 Mbit/s
CAN-HG Transceiver Filtering (Canis Labs)	Low-level protocol abuse	Blocks malformed or glitch frames in hardware, before ECU logic	No payload check; extra hardware cost; proprietary
Network Segmentation + Secure Gateways	Lateral movement control	Common in production; separates domains (infotainment vs. drivetrain)	Gateway compromise ruins isolation; harder diagnostics and updates
Intrusion Detection Systems (IDS)	Anomaly and spoof detection	Non-intrusive; real-time alerts; works with existing traffic	Possible false alarms; reactive only; needs ongoing tuning

†CryptoCAN uses a software HSM (SHE style) to create authentication tags [30].

Take-away. A balanced plan is to sign a few safety-critical IDs, fit PHY-level filters to stop protocol abuse, and run an anomaly-based IDS for live insight. CANvas can replay and stress-test exactly these combinations, letting designers measure residual risk *before* hardware is finalised.
No single defence is enough, yet together they raise the skill and cost needed for both casual theft and advanced intrusion.

Section 2.3 surveys existing automotive-security tools and showcase the identified gaps.

2.3 Landscape of Automotive-Security Tooling

A decade of car-hacking workshops and research papers has produced a rich—but fragmented—ecosystem of security tools. At one extreme sit proprietary validation suites such as *Vector CANoe*; at the other, lightweight open sources like *can-utils*. Somewhere in-between are research frameworks (Simutack, Laufenberg *etal.*) and attack-centric toolkits such as *CANToolz*. This section reviews the state of the art along three questions:

a) **What domains are modelled ?** (in-vehicle buses, sensors, V2X links, cloud back-ends)

b) **How realistic is the behaviour ?** (timing accuracy, physics-based dynamics, error handling)

c) **How far can we go from "observe" to "exploit" ?** (diagnostics only, or full packet-crafting and fuzzing)

2.3.1 Review of Existing Tools and Frameworks

Industrial test benches (Vector CANoe/CANalyzer [35]). Vector tools are the industry standard for automotive network conformance and regression testing. Specifically, CANoe simulates multiple ECUs with cycle-accurate timing. CANalyzer, on the other hand, focuses on traffic analysis, stimulation, and diagnostics without built-in simulation of vehicle dynamics. However, both tools are commercial, involve high four- to five-figure licence costs, and offer limited cybersecurity testing—primarily restricted to scripted message injections without integrated attack libraries.

Multi-domain research simulators (Simutack [36]). The Technical University of Munich's open-source *Simutack* combines OMNeT++, CARLA, and SUMO to simulate sensors, Vehicle-to-Everything (V2X), and in-vehicle network traffic. It is ideal for generating datasets for Intrusion Detection Systems (IDS) research, but its complexity and GPU requirements make it challenging for interactive or real-time fuzz-testing scenarios.

CAN-only traffic generators (Laufenberg et al. [37]). This OMNeT++-based framework generates synthetic CAN logs from a defined ECU network, driving scenarios, and environmental conditions. It includes approximately 45 predefined attack templates (e.g., spoofing, flooding, fuzzing) to tag malicious frames. The resulting high-fidelity traces are suited for offline IDS model training. However, the tool currently provides no interactive dashboard, live playback, or easily extendable modular attack scripts. The authors have not publicly released their source code repository at the time of writing.

Red-team toolkits (CANToolz [38]). CANToolz is an open-source, SocketCAN-based toolkit designed for real-time CAN message injection, fuzzing, and black-box testing. It effectively supports live vehicle experiments, virtual CAN (vcan) networks, and USB CAN adapters. However, it does not simulate vehicle dynamics; consequently, gauges and visual outputs remain static unless driven by a replayed CAN log.

Instrument-cluster visualisers (ICSim [8]). ICSim offers an SDL-based instrument-cluster GUI displaying a speedometer, tachometer, warning lights, and door status upon receiving appropriate CAN frames over a virtual CAN interface. It is excellent for classroom demonstrations and introductory CAN bus workshops, providing basic keyboard-driven message injections. However, its scope is limited to replaying a short static trace without dynamic vehicle simulation, background traffic, scriptable attacks, or IDS evaluation support. Additionally, message layouts are hard-coded, as there is no support for importing standard CAN database (DBC) files.

2.3.2 Comparative Matrix

Table 2.5: Feature comparison of representative automotive-security tools.

Tool	**Domain scope**	**Attack lib.**	**Vehicle dynamics**	**Accessibility**	**Live GUI**	**Typical use case**
CANoe & CANalyzer [35]	CAN/LIN/ ETH + diagnostics	unknown	yes	Proprietary	Rich	OEM conformance
Simutack [36]	Sensors + V2X + CAN	scripted	full (CARLA)	Open-source	None (visualization via CARLA)	IDS dataset generation
Laufenberg [37]	CAN only	BTM models	no	Research code on request (no public repo at time of writing)	none	Synthetic CAN log generation for offline IDS training
CANToolz [38]	CAN bus	(modular)	no	Open-source	CLI	Black-box fuzzing
ICSim [8]	CAN + dash	none	minimal	Open-source	Cluster gfx	Teaching / demos

2.3.3 Identified Gaps

The table 2.5 exposes four persistent shortcomings.

- **Realistic *and* interactive.** Simutack and CANoe provide physics realism but little red-team agility; CANToolz provides agility but no context.
- **Cost and accessibility.** most academic labs cannot afford commercial CANoe licences, limiting reproducibility and slowing research.
- **Repeatable security benchmarks.** many IDS studies rely on private traces, making it hard to compare detection rates across papers or reproduce results.
- **Extensibility.** As the threat landscape evolves each year, there is a growing need for a platform that researchers can easily extend without modifying the underlying core architecture.

Conclusion

This chapter has explored the evolution of the Controller Area Network (CAN) from its fundamental frame structure to its role within today's increasingly complex, software-defined vehicles. The inherent robustness of CAN, including mechanisms like arbitration and error handling, also introduces vulnerabilities that attackers can exploit. Real-world threats span from simple, low-cost physical manipulations to sophisticated remote attacks

such as over-the-air ransomware, emphasizing the need to consider both physical and remote attack vectors comprehensively. Existing tools in automotive cybersecurity typically focus on specific aspects without fully integrating realistic vehicle dynamics, live CAN bus interactions, and open-source flexibility. These limitations highlight the necessity for a simulation platform that integrates these elements cohesively, a need addressed by the CANvas platform detailed in the subsequent chapter.

Chapter 3
Project Objectives and Requirements

This chapter defines the mission, objectives, and functional scope of the **CANvas** platform. It builds directly upon the vulnerabilities, limitations, and tooling gaps outlined in Chapter 2, transitioning from problem space to design response. The analysis in the previous chapter underscored the fragmentation between high-fidelity automotive simulators and agile cybersecurity frameworks. CANvas is introduced as a solution that unifies both aspects in a single, accessible environment.

Chapter overview. Section 3.1 establishes the project's target by contextualizing the identified tooling gap; Section 3.2 derives formal functional requirements; Section 3.3 defines the boundaries and constraints of the work; and Section 3.4 presents the top-level system architecture.

3.1 Defining the Project's Target

As discussed in Chapter 2, the current landscape of automotive cybersecurity tools is fragmented. Some tools offer simulation of vehicle sensors but provide limited support for active threat modeling or real-time experimentation. Others enable flexible attack scripting and diagnostic testing, but do so in isolation from realistic environmental and vehicular dynamics.

3.1.1 Existing Solutions and Their Limitations

This gap becomes evident when reviewing representative tools from both ends of the spectrum:

Vector CANoe: A commercial-grade simulation and validation suite supporting extensive protocol testing and HIL integration. However, it lacks integrated cybersecurity modules and remains cost-prohibitive for most academic institutions.

SimuTack: Offers a rich multi-domain simulation (including sensors, traffic, and V2X), ideal for dataset generation. However, it involves high complexity and offers limited agility in cybersecurity experimentation.

CANToolz: Highly scriptable and powerful for live CAN manipulation, particularly in penetration testing contexts. Nonetheless, it provides no environmental realism or physical-state feedback.

ICSim: An educational tool visualizing CAN messages on a dashboard cluster. It is limited to static trace playback and lacks support for dynamic inputs or attack modules.

While each of these tools addresses specific needs, none offer a unified approach combining real-time simulation, realistic stateful vehicle modeling, and modular attack experimentation within a cohesive open-source framework.

3.1.2 CANvas: Bridging the Gap

There is a clear absence of an open-source platform that combines context-aware vehicle simulation with dynamic and modular cybersecurity testing. This gap is not only technical but also practical, limiting the ability of researchers, students, and industry testers to perform reproducible, real-time experiments.

CANvas is designed to fill this need by offering a unified platform that integrates vehicle-state modeling, CAN message generation, and modular cybersecurity interaction. Its key capabilities include:

- **Live, state-driven CAN generation** — Environmental inputs such as throttle, steering angle, and road conditions produce signal-accurate CAN traffic in real time.
- **Modular experimentation support** — Attack, defense, and diagnostic modules can be integrated dynamically without modifying the core system.

Through this combination, CANvas enables use cases ranging from classroom instruction to penetration testing, IDS evaluation, and protocol fuzzing, offering a flexible environment for automotive cybersecurity research and training.

3.2 Detailed Project Requirements

To ensure clarity, we explicitly state and categorize the requirements governing CANvas's development.

3.2.1 Functional Requirements (FR)

FR1: Real-time generation of standard CAN bus traffic, with planned incremental support for CAN FD extended frames (up to 64-byte payloads and extended identifiers).

FR2: Integration of dynamic vehicle-state simulation (speed, brake, steering angle, etc.).

FR3: Native import and processing of DBC files for signal accuracy.

FR4: Extensible, modular plugin system for security testing scenarios (spoofing, fuzzing, diagnostics).

FR5: Export capability to support IDS development and evaluation.

3.2.2 Non-Functional Requirements (NFR)

NFR1: Compatibility and ease of deployment on Linux systems leveraging SocketCAN interfaces.

NFR2: Comprehensive, straightforward installation and detailed documentation ensuring ease-of-use.

NFR3: Scenario reproducibility using structured JSON scripts for repeatable experiments.

NFR4: Interactive, visual feedback (e.g., dashboard, logging window) during active experiments to support the analysis.

3.2.3 Coverage by Existing Solutions

Some functional requirements partially overlap with existing tools:

- CAN bus traffic generation (FR1) is addressed by Vector CANoe. Same for CANToolz but it lacks environmental context
- Basic visualization of vehicle states (FR2) is minimally supported by ICSim.

3.2.4 Unmet Requirements Identified

Key critical requirements currently unfulfilled by existing solutions include:

- Direct synchronization between vehicle dynamics and generated CAN bus traffic (FR2).
- User-friendly, reproducible, and scalable experimental setup for cybersecurity testing (NFR3).
- Fully modular, open-source cybersecurity experimentation framework (FR4, NFR1).

3.3 Project Scope and Boundaries

Clearly outlining what CANvas intends and explicitly does not intend to achieve ensures realistic project management and stakeholder clarity.

3.3.1 Mission Statement

> *CANvas aims to provide an accessible, reproducible, and practical open-source platform that integrates high-fidelity CAN-bus vehicle simulation with advanced, modular cybersecurity experimentation—effectively bridging the gap between complex vehicle simulators and agile red-team toolkits.*

3.3.2 Explicitly Out-of-Scope Features

To avoid ambiguity and set clear boundaries, the following elements are explicitly outside CANvas's initial scope:

- Proprietary ECU firmware emulation beyond basic standardized UDS services.
- Complex integration with high-end physics simulation platforms (CarSim [39], IPG Carmaker).
- Support for closed or vendor-specific automotive protocols and extensions not publicly documented.
- Comprehensive Vehicle-to-Everything (V2X) communication stacks; this current version of the tool and our current mission is solely focused on intra-vehicle CAN networks.

3.4 CANvas Top-Level Design

The CANvas platform integrates multiple modular components specifically designed to bridge realistic automotive simulations with dynamic cybersecurity experimentation. To provide clarity from the outset, Figure 3.1 illustrates the top-level architecture, component interactions, and overall data flow within the CANvas platform.

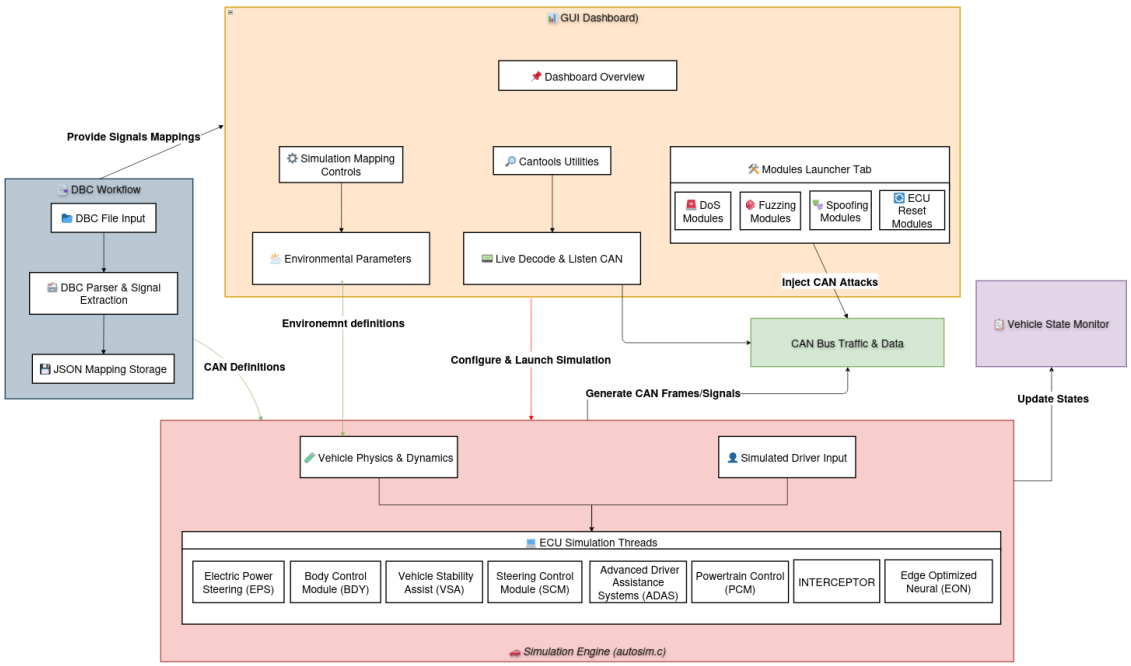

Figure 3.1: CANvas Top-Level Architecture and Data Flow

3.4.1 Overall Workflow and Intended Usage

The intended workflow of CANvas can be summarized as follows:

1. Users begin by loading standard DBC files via the GUI, which are processed through the **DBC Workflow**. This workflow, implemented in the `dbc_parser` module of our project (based on [9]), automatically parses the DBC definitions and produces structured JSON mapping files that describe CAN signals, ECUs, and messages in a clear and reproducible format.

2. With the generated mappings, users configure simulation scenarios within the **Simulation & Mapping Controls**, specifying environmental conditions (e.g., weather, road friction) and traffic density through intuitive menus in the GUI dashboard.

3. Users launch the simulation engine (`autosim.c`), which continuously computes realistic vehicle dynamics and generates accurate CAN messages reflecting these simulated states. The simulation can incorporate randomized driver input and environmental influences for greater realism. These internal dynamics and their corresponding ECU behaviors are discussed in detail in Chapter 4.

4. During simulations, users can dynamically activate and control cybersecurity attack modules (e.g., DoS, fuzzing, spoofing, ECU resets) directly from the GUI, injecting malicious frames into the CAN traffic stream.

5. CANvas continuously decodes live CAN traffic via integrated **Cantools Utilities**, displaying real-time updates and vehicle states on the GUI dashboard, thus facilitating near-real-time visibility of simulation results and attack impacts.

6. Finally, users export detailed logs which are crucial for further analysis, Intrusion Detection System (IDS) benchmarking, and reproducibility of cybersecurity experiments.

To support these capabilities, CANvas is composed of several core components, each tailored for a specific role in the system. The following sections introduce these components in detail, outlining how they interact and contribute to the platform's overall functionality.

3.4.2 Graphical User Interface (GUI) Dashboard

The GUI dashboard (`CANvas.py`) serves as the primary user interface, centrally coordinating all aspects of CANvas operations. As shown in Figure 3.1, it connects directly with the DBC workflow, simulation control layer, live decoding utilities, and dynamic attack modules—allowing users to manage and monitor the simulation environment in real time.

The dashboard is composed of distinct yet integrated tabs:

- **Simulation Mapping Controls:** Allows users to configure simulation parameters, including environmental conditions (e.g., road friction, weather) and scenario types, by directly interacting with the vehicle dynamics engine.
- **Modules Launcher Tab:** Exposes all supported cybersecurity attack modules (DoS, fuzzing, spoofing, ECU resets), which can be injected into the live CAN traffic stream dynamically.
- **Cantools Utilities:** Provides live CAN decoding during runtime, translating raw CAN frames using pre-loaded DBC mappings. This allows users to visualize bus traffic in human-readable form and immediately assess attack effects.
- **Dashboard Overview:** Consolidates system status and runtime metrics, offering high-level feedback on vehicle states, simulation timing, and attack activity.

This modular structure ensures fluid interaction with both simulation and cybersecurity components, enabling rapid scenario prototyping, debugging, and experimentation—all from a single unified interface.

3.4.3 Simulation Interface: Physics Engine and Vehicle State Management

CANvas's simulation interface, primarily implemented in `autosim.c`, provides high-fidelity, real-time vehicle dynamics essential for generating accurate CAN bus traffic. The interface ensures realistic synchronization between simulated physical states and CAN bus messages, enhancing the authenticity of cybersecurity scenarios and facilitating meaningful attack impact analysis.

From *ICSim* to the CH-Workshop fork, then to CANvas. The original *ICSim* demo offered a static dashboard and a few keyboard-driven CAN frames. Philippe Blot's "CH-Workshop" fork already pushed that baseline further by adding UDS/OBD-II command handling, embedding a toy diagnostic tool and introducing a scoring system for capture-the-flag workshops. CANvas takes the baton and layers a full physics engine plus multi-ECU CAN generation on top of that fork, turning a learning game into a cyber-physical research testbed :

- **Realistic Vehicle Dynamics Simulation (`autosim.c`):** A robust, lightweight physics engine simulates essential vehicular dynamics including acceleration, braking responsiveness, steering angle adjustments, and interactions with environmental

parameters like gradient, wind resistance, and road conditions. The simulation operates continuously in real-time (targeting a 50 Hz refresh rate), updating vehicle parameters to closely emulate authentic driving conditions.

- **Interactive Vehicle State GUI** Provides an intuitive, real-time interactive menu, allowing detailed monitoring of vehicle state parameters (speed, acceleration, engine temperature, gear, throttle levels, door states, and more). Users can instantly observe these states, ensuring comprehensive exploration of vehicle behaviors under varied simulated scenarios.

- **Synchronization with CAN Message Generation:** Vehicle state parameters directly inform the generation of CAN messages, creating tight synchronization between the simulation layer and CAN bus activity. This synchronization ensures the authenticity of cybersecurity testing environments, allowing accurate assessment of attacks' impact on realistic vehicle responses.

The combined physics engine and interactive state management deliver a comprehensive, realistic vehicle testbed essential for cybersecurity scenario validation and IDS benchmarking.

3.4.4 Attack Interface: Dynamic Module Loader

The attack interface is exposed through the GUI's **Modules Launcher Tab**, allowing users to inject malicious CAN frames directly into the ongoing simulation via a modular plug-in architecture. As visualized in the architecture (Fig. 3.1), the launcher operates in real-time, injecting attacks into the CAN bus traffic stream and immediately reflecting effects across the vehicle state and decoded data views.

Key capabilities of the attack interface include:

- **Modular Activation:** Users can toggle specific attack types such as DoS floods, signal spoofing, fuzzing, and ECU reset modules. Each module is activated through a graphical dialog that exposes its parameters and run-time controls.

- **Tight Simulation Coupling:** The injected frames affect running vehicle state threads (e.g., EPS, PCM, ADAS), enabling realistic impact analysis. These changes are immediately visible via the GUI dashboard and vehicle state monitor.

- **Live Feedback Loop:** Each module's output is logged and rendered in color-coded GUI panes, showing stdout/stderr streams and live frame statistics, helping researchers monitor stability, reaction timing, and countermeasure effectiveness.

This integration transforms CANvas from a passive simulator into an active cyber-physical testing ground, where users can explore, iterate, and refine real-world threat scenarios. A complete description of all implemented attacks appears in Chapter 5.

3.4.5 DBC Workflow and CAN Message Management

A dedicated *DBC workflow* transforms raw specification files into experiment-ready artefacts, eliminating the traditional overhead of CAN-bus pre-processing. The process unfolds in four tightly-coupled stages:

1. **Automated Ingestion.** When a DBC file is selected, CANvas parses it in the background—extracting messages, signal layouts, scaling factors, and ECU ownership. Optional filter keywords give users fine-grained control over what is imported, while the parser enforces naming and range checks to prevent silent inconsistencies.

2. **Interactive Signal Resolution.** Any ambiguity (e.g. duplicate IDs or variant-specific signals) is surfaced through wizard-style dialogs. The user can accept sensible defaults or override individual fields, ensuring every signal is unambiguously defined before simulation begins.

3. **Persistent JSON Mapping.** The validated model is serialised to a compact JSON file and stored in `data/mappings/`. These files can be re-loaded across sessions, shared with collaborators, and version-controlled, making experiments perfectly repeatable.

4. **Live Decoding and Analytics.** During a run, the saved mapping feeds the integrated *cantools* / *python-can* stack, allowing the GUI to translate raw frames into structured signal read-outs for immediate verification. Researchers can immediately verify that injected or spoofed traffic conforms—or deliberately deviates—from the expected format.

By collapsing what is usually a fragmented tool chain into a single, interactive pipeline, the CANvas DBC workflow removes tedious preprocessing, minimises mapping errors, and delivers a solid foundation for large-scale automotive cybersecurity experiments.

3.4.6 Underlying Communication Infrastructure

CANvas leverages Linux's SocketCAN to ensure rapid exchange of vehicle and environmental states among modules. Additionally, structured logging enables the automated labeling of simulation outputs, simplifying the creation of benchmark datasets for Intrusion Detection Systems.

In summary, the CANvas architecture integrates clearly defined and interactive components into a cohesive simulation and cybersecurity experimentation platform. Its intuitive GUI dashboard facilitates dynamic user control, while the modular backend ensures accurate vehicle dynamics, realistic cybersecurity threats, and reproducible experimental results. This structured approach supports both research and educational use, emphasizing flexibility, realism, and ease-of-use.

Chapter Summary

In this chapter, we have explicitly established the mission, objectives, and detailed requirements for the **CANvas** project, highlighting the specific gaps identified in existing automotive cybersecurity tools. We identified key community needs, reviewed limitations of existing solutions, and introduced CANvas as a unified, open-source solution integrating vehicle dynamics simulation with cybersecurity experimentation.

In the next chapter, we will delve deeper into the specifics of ECU communications, detailing the internal logic and physical modeling behind our vehicle simulations.

Chapter 4
Implementation

Having defined *what* CANvas must do, we now explain *how* it does it. This chapter descends from the top-level software architecture to the lowest-level physics engine in five progressively narrower layers:

1. the multi-threaded **simulator architecture** that orchestrates Electronic Control Units (ECUs);
2. the **DBC intake pipeline**, which turns proprietary CAN databases into a clean JSON mapping;
3. the detailed **ECU logic reference** for all eight simulated controllers;
4. the vehicle-dynamics core (§ 4.4)
5. the sensor-realism layer (§ 4.5) that injects noise and jitter.

While the first two layers provide a high-level operational overview of CANvas, the subsequent sections delve into low-level architectural and implementation details. These are intended primarily for readers seeking a deeper understanding of the system internals or aiming to extend the platform's functionality.

4.1 Simulator Architecture and ECU Threads

4.1.1 Multithreaded Execution Model

Each Electronic Control Unit (ECU) in CANvas runs as an independent POSIX thread, simulating a modular, real-time embedded system. The simulation includes an optional physics thread for vehicle dynamics when multicore systems are available. CANvas relies on Linux's native `SocketCAN` interface to emulate realistic CAN bus communication.

Each ECU thread follows a periodic execution cycle:

1. Waits for incoming CAN frames or timer events using `poll()`;
2. Reads matching CAN frames via `read()`;
3. Executes ECU-specific logic based on current vehicle state and inputs;
4. Sends output frames with `write()` to the virtual CAN bus;
5. Sleeps until the next scheduled cycle using `timerfd`.

The table below summarizes the main ECU threads used in CANvas, along with their execution intervals and core responsibilities.

ECU	Period	Real-world Role	Key Function
ADAS	100 ms	Driver Assistance (AEB / ACC / LKAS)	Controls safe distance, braking, and lane centering.
PCM	100 ms	Powertrain Control	Regulates engine power, gears, and vehicle speed.
EPS	100 ms	Electric Steering	Assists driver steering based on speed and road conditions.
VSA	20 ms	Stability Control	Helps prevent skidding and keeps the vehicle stable.
SCM	20 ms	Steering Column Controls	Handles user inputs like blinkers, lights, and wipers.
EON	100 ms	Autopilot Interface	Sends automatic gas and steering commands.
Interceptor	100 ms	Pedal Signal Filter	Monitors and safely forwards gas pedal signals.
BDY	200 ms	Body Control	Tracks doors, seatbelts, and basic safety systems.

Table 4.1: Simplified overview of ECU threads in CANvas with timing and core functions.

With the core thread architecture established, we now turn to the environmental model

4.1.2 Environmental Model

In CANvas, a central `environment` structure is used to represent real-world driving conditions. This structure is shared across all ECUs and is initialized at the beginning of each simulation, either through the graphical interface or via scenario files. Once a simulation starts, the values remain fixed to ensure experiments are reproducible. However, scripted scenarios can update the environment between simulation laps.

The environment model includes the following key parameters:

- **Weather** — Options include *clear*, *rain*, *snow*, or *fog*. These affect tire grip and the reliability of sensors such as radar and cameras. For example, rain may reduce traction and visibility, which in turn triggers more frequent interventions from safety systems like stability control or emergency braking.
- **Time of Day** — Set to *day*, *dusk*, *night*, or *dawn*. This influences lighting behavior

and camera sensitivity. In darker conditions, systems like automatic headlights or lane detection adjust to maintain visibility and safety.

- **Road Slope (Gradient)** — A value between -0.05 and $+0.05$ represents downhill or uphill inclines. This parameter affects how the engine and brakes respond. Uphill slopes increase engine load, while downhill conditions require stronger braking and may alter cruise control behavior.

- **Temperature** — Ranges from $-40°C$ to $+60°C$. Extreme temperatures influence battery efficiency, sensor accuracy, and engine performance. For instance, colder temperatures can make sensors less reliable.

- **Traffic Density** — A value from 0 (empty road) to 1 (high traffic). This affects driver assistance systems like adaptive cruise control, which will maintain larger gaps in busy traffic, and high-beam logic, which becomes more conservative to avoid dazzling nearby drivers.

These environmental factors are critical to simulating realistic vehicle behavior. They allow researchers to study how changes in weather, lighting, terrain, and traffic affect both control systems and safety features. For instance, a foggy night on a steep hill with high traffic will challenge the vehicle in multiple ways—testing the robustness of adaptive cruise, emergency braking, and visibility systems simultaneously.

Further details on how each ECU responds to environmental conditions are presented in Section 4.3. Before that, we will explore how ECUs communicate with each other through the virtual CAN bus.

4.1.3 Inter-ECU Data Flow

Figure 4.1 provides a visual overview of how the eight ECU threads in CANvas interact through CAN message exchanges, corresponding to those listed in Table 4.1. In the diagram, green rectangles represent the software-based ECUs, while blue rounded boxes depict the CAN frames being transmitted across the virtual bus. Black arrows indicate the incoming frames each ECU consumes, and red arrows represent the messages it generates in response. This flow reflects the **default communication logic** defined by the simulation engine. However, the behavior and signal structure of the ECUs are **flexible** and can be **customized** by the user—particularly through modifications made to the parsed DBC files during the setup phase.

With the execution model in place we next explain *how CANvas learns the language* spoken on the bus—its DBC parsing and normalisation pipeline.

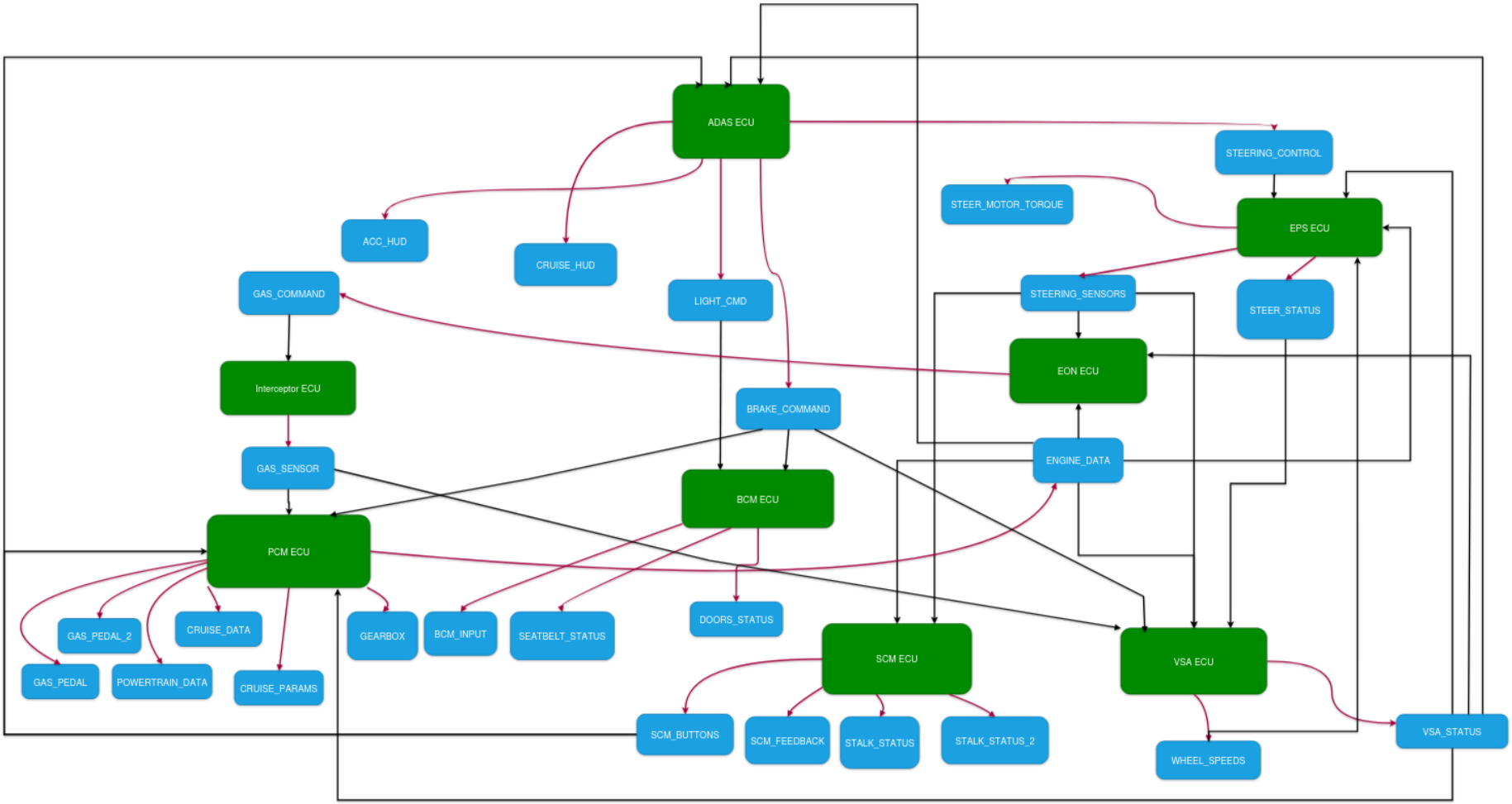

Figure 4.1: Run-time data flow between ECU threads (auto-generated).

4.2 DBC Parser and Signal Normalization

Our automotive cybersecurity simulation fundamentally depends on accurately interpreting vehicle communication standards described within CAN Database (DBC) files. These files contain structured definitions of how ECUs (Electronic Control Units) exchange messages and signals on a vehicle's CAN bus. Given their critical role, CANvas implements a flexible Python-based parser, `dbc_parser.py`, to reliably extract and normalize information from these files.

4.2.1 Parsing and Normalization Process

During development, we encountered several practical challenges when working with DBC files. To ensure clean and consistent parsing, we introduced the following solutions:

- **Pre-processing and Cleanup:** Many DBC files found in the wild include proprietary extensions, irregular formatting, or extra characters that can confuse standard parsers. To handle this, our parser begins by cleaning up these inconsistencies—removing things like unusual keywords, duplicated punctuation, or non-standard CAN identifiers using targeted regular expressions. This step helps avoid errors and improves the reliability of the rest of the parsing process.

- **Signal Name Normalization:** Different manufacturers use different naming styles for signals, which can make things confusing. To fix this, CANvas automatically converts varied naming formats (like `Eng_RPM`, `EngineRPM`, or `RPM_ENG`) into a single consistent format (e.g., `engine_rpm`). When there's ambiguity—such as multiple names mapping to the same standard form—the system asks the user to choose via a simple pop-up dialog, keeping things both accurate and user-friendly.

4.2.2 Understanding DBC File Structure

To facilitate clear understanding, Figure 4.2 illustrates the basic syntax of DBC files. Messages are defined by lines starting with `BO_`, indicating the CAN ID, sender ECU, and data length. Signals within messages are introduced by `SG_`, specifying details like bit position, length, scale, offset, units, and the receiving ECU. Our parser systematically extracts these parameters directly, as demonstrated in the annotated figure.

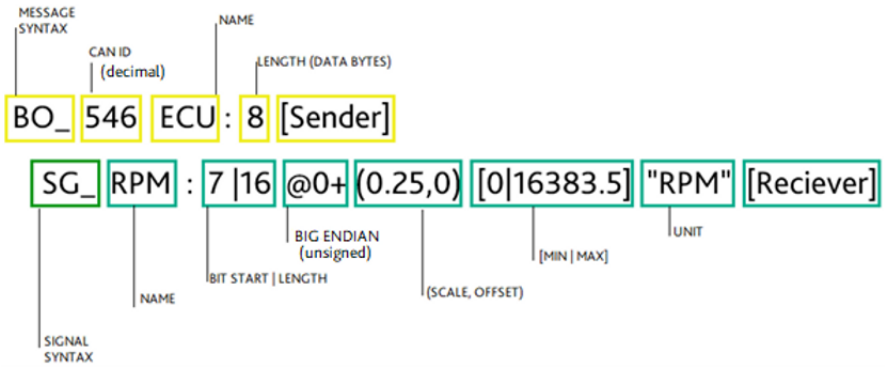

Figure 4.2: DBC file structure example showing message (`BO_`) and signal (`SG_`) definitions [4].

4.2.3 Creating ECU Communication Maps

After successful parsing and normalization, the final structured mapping is stored as a compact JSON file, creating a single authoritative reference point for all CANvas components. This JSON structure follows a consistent organization pattern as exemplified below:

```
{
    "ecu_mapping": {
        "ADAS": {
            "BRAKE_COMMAND": {
                "message_name": "BRAKE_COMMAND",
                "can_id": 506,
                "is_fd": false,
                "frame_length": "8",
                "signals": [
                    {
                        "name": "COMPUTER_BRAKE",
                        "offset": 0.0,
                        "scale": 1.0
                    },
                    // Additional signals...
                ]
            },
            // Additional messages...
        },
        // Additional ECUs...
    },
    "environment": {
        "weather": "sunny",
        "time_of_day": "day",
        "road_condition": "dry",
        "gradient": 0.0,
        // Additional environmental parameters...
    }
}
```

The file contains two main sections: `ecu_mapping` and `environment`. The `ecu_mapping` section organizes CAN messages by ECU type (ADAS, BDY, EON, etc.), with each message defined by its identifier, frame properties, and detailed signal parameters. The `environment` section stores simulation context variables that affect ECU behaviors, such as weather conditions and road characteristics. This structured approach enables consistent signal interpretation across simulation components while maintaining environmental context for realistic cyber-physical interactions.

At this point, we have established how CANvas prepares and standardizes CAN signals. Next, we describe how these normalized signals drive detailed ECU simulations.

4.3 ECU Functional Details

Having parsed the CAN communication layout, CANvas maps this structure to dedicated ECU logic threads, each simulating realistic behaviors and interactions based on sensor input and inter-ECU messaging.

4.3.1 Advanced Driver-Assistance Systems (ADAS)

The ADAS thread runs at a steady **10 Hz** (100 ms), mirroring the timing budget found in production vehicles for longitudinal and lateral assistance.

CAN exchange at a glance: Table 4.2 lists every frame the PCM consumes or publishes in the default demo mapping; the user may remap the IDs in the JSON generated by the DBC parser.

Frame (name / key signals)	Default CAN IDs (hex)	Dir.	Purpose inside ADAS
ENGINE_DATA	0x316	→	vehicle speed
GAS_PEDAL	0x30C	→	gas pedal position
VSA_STATUS	0x18F	→	brake pressure
SCM_BUTTONS	0x1A2	→	cruise control buttons
CRUISE_DATA	0x324	→	cruise target speed and state
STEERING_CONTROL	0x0E4	←	steering torque request
BRAKE_COMMAND	0x1FA	←	computer brake control
ACC_HUD	0x17C	←	ACC status and warnings
HIGHBEAM_CONTROL	0x35E	←	high beam control

Table 4.2: Key CAN frames processed by the ADAS ECU (Dir. → = consumed by ADAS, ← = produced by ADAS).

Overview on the internal logic

- **Adaptive Cruise Control (ACC)** PI-controller holds the target speed or, if a slower lead-vehicle speed is supplied by the radar stub, maintains a dynamic spacing of 2.0–2.5 s. Fallback to simple speed hold when `radar_obstructed=1`. Driver override reduces braking when gas pedal $> 30\%$.

- **Autonomous Emergency Braking (AEB)** Triggers when speed > 50 km/h and traffic density > 0.8, or when speed > 60 km/h in bad weather. Linearly ramps `computer_brake` up to 100 %. Road conditions (wet, icy, gravel) scale the braking request.

- **Lane-Keeping Assist (LKAS)** Adjusts steering torque based on wind conditions and EPS state. Produces a capped steering-torque request (max 3.84 Nm in demo).

- **Auto High-Beam (AHB)** Enables high beams when *night* & traffic density < 0.3, else keeps low beams. Syncs with SCM state for consistency.

4.3.2 Power-train Control Module (PCM)

The PCM thread acts as the vehicle's engine + gearbox brain, converting driver pedals and ADAS overrides into longitudinal torque, automatic gear shifts and cruise-control updates. The PCM thread also ticks at **100 ms**.

CAN exchange at a glance. Table 4.3 lists every frame the PCM consumes or publishes in the default demo mapping; the user may remap the IDs in the JSON generated by the DBC parser.

Frame (name / key signals)	Default CAN IDs (hex)	Dir.	Role in PCM logic
GAS_SENSOR (dual ADC channels)	0x1A0	→	Driver throttle %, enable flag
VSA_STATUS (ESP feedback)	0x18F	→	Brake pressure, ESP status, stand-still
SCM_BUTTONS (wheel-stalk)	0x1A2	→	Cruise-control buttons (SET/RES/CANCEL)
ENGINE_DATA (speed, RPM, odo)	0x316	←	Vehicle speed, engine RPM, odometer
GAS_PEDAL (normalized)	0x30C	←	Normalized gas pedal position
GAS_PEDAL_2 (torque)	0x304	←	Engine torque estimate and request
POWERTRAIN_DATA (status)	0x17C	←	Pedal %, RPM, brake flags
CRUISE_DATA (speed, state)	0x324	←	Cruise control speed and state
GEARBOX (shifter, gear)	0x33A	←	Transmission state and gear

Table 4.3: Key CAN frames processed by the PCM ECU (Dir. → = consumed, ← = produced).

Overview on the internal logic At every tick the PCM

1. merges *driver commands* (‘gas_pos‘, brake pedal) with *ADAS overrides* (‘brake_req‘, cruise set-speed);
2. integrates a first-order **vehicle-mass model** (drag, rolling losses, road gradient) → speed;
3. back-projects the resulting acceleration to estimate engine torque and selects an automatic gear using speed-based up/down thresholds (`UP_THRESH/DN_THRESH`);
4. updates **cruise-control** state via a small proportional controller when buttons are pressed;
5. simulates engine temperature ($\dot{T} = q_{\text{heat}} - q_{\text{cool}}$) for later fan / fail-safe work; and
6. transmits the seven frames in Table 4.3, each carrying a rolling counter and the 4-bit Honda checksum $\left(\sum \text{data} + ID + len\right) \bmod 16$.

4.3.3 Body-Control Module (BDY)

The body ECU is a “comfort / low-speed safety” node: it aggregates door-ajar switches, seat-belt latches, air-bag lamp status and brake-lamp activation, then republishes this data for the cluster, ADAS and gateway ECUs. Its thread fires every **200 ms**.

CAN exchange at a glance. Table 4.4 lists the default frames; IDs come from the default configuration and may be remapped in the JSON produced by the parser.

Frame (name — key signals)	**Default CAN IDs (hex)**	**Dir.**	**Role inside BDY logic**
BRAKE_COMMAND (ADAS/AEB)	0x1FA	→	Reads bit 7 of byte 4 to mirror brake-lamp request
BCM_INPUT (door, seat-belt, lamps)	0x190	←	One-byte bitmap of FL/FR/RL/RR doors, trunk and brake lamps
SEATBELT_STATUS (driver/pass)	0x305	←	Latched / un-latched flags + integrity counter / checksum
DOORS_STATUS (all five doors)	0x405	←	Door-ajar + trunk status for cluster animation

Table 4.4: Key CAN frames processed by the BDY ECU (Dir. → = consumed, ← = produced).

Internal logic: When starting, the BDY thread executes a tiny "driver script" (doors close at 5 s, seat-belt latches at 7 s, auto-lock at 7.5 s) so that simulation demos look alive; Then each 200 ms cycle it

1. drains the CAN RX queue and copies `brake_lights` from `BRAKE_COMMAND`;

2. transmits

   - **BCM_INPUT** — raw bitmap in byte 0 (doors, trunk, brake lamps) and byte 1 (seatbelts, airbag), no counter / checksum,
   - **SEATBELT_STATUS** — driver/passenger latched flags in byte 0, airbag status in byte 1, with 4-bit counter and checksum,
   - **DOORS_STATUS** — door states in byte 0 (RL, FR, FL, RR, trunk), with 4-bit counter and checksum;

3. logs any CAN-TX failure and continues (the body bus is non-critical).

4.3.4 EON "Open-Pilot" Controller

The *EON* ECU is a minimal cruise-only autopilot that requests engine torque but *never* touches the brake; this makes it an ideal victim for spoofed speed or steering data. The task runs at 100 ms.

CAN exchange at a glance. Table 4.5 shows the default frame set. Only one transmit frame is generated, yet it already demonstrates Honda's redundant-channel pattern (dual 16-bit pedals, 4-bit rolling counter, 1-byte checksum).

Frame (name — key signals)	Default CAN IDs (hex)	Dir.	Role in EON logic
ENGINE_DATA (speed)	0x316	→	True longitudinal speed (0.01 km/h LSB)
STEERING_SENSORS (angle, rate)	0x17E	→	Current wheel angle and yaw rate
VSA_STATUS (brake, ESP flags)	0x18F	→	Absolute brake level and wheel-slip bits
GAS_COMMAND (pedal 1+2, enable)	0x1A1	←	Redundant gas request

Table 4.5: Key CAN frames processed by the EON ECU (Dir. → = consumed, ← = produced).

Internal logic (summary). Each cycle the EON thread performs the following steps:

1. **Pedal profile.** A synthetic driver signal ramps linearly from $0 \rightarrow 1$ over 5 s, then back to 0 — continuously exercising the power-train.

2. **Environment scaling.** The raw pedal value is multiplied by three context factors (altitude, road gradient, ambient temperature):

$$\text{cmd} = p \times f_{\text{alt}}(h) \times f_{\text{grad}}(\gamma) \times f_T(\vartheta)$$

   ensuring high mountains, steep climbs or very hot/cold weather reduce the requested torque.

3. **Clamping slewrate.** The command is limited to the physical range and never changes faster than ± 0.5 per tick.

4. **Frame packing.** Two 16-bit channels (`raw1/raw2`) carry the request with distinct scaling, followed by:
   - bit 7 of byte 4 — "enable" flag,
   - bits 3-6 — 4-bit rolling counter,
   - byte 5 — Honda checksum $\big(\sum \text{data} + ID + \text{len}\big)\&0xFF$.

5. **Safety mirror.** The final pedal request is copied into `eon_state.gas_command` for live HUD display.

Because EON's output is a single high-authority frame, attackers can experiment with *speed spoofing* or *steering-sensor tampering* and observe how the autopilot either under- or over-shoots its target torque — a compact yet representative illustration of cyber-physical side-effects.

4.3.5 Electric-Power Steering (EPS)

The EPS controller emulates a modern, column-mounted assist unit. Its task loop fires every **100 ms** and blends

1. the driver's own steering input (randomised yet smooth state-machine),

2. ADAS torque requests arriving on `STEERING_CONTROL`, and

3. speed-dependent electric boost plus subtle environment bias (wind, gradient, temperature),

into a single motor-torque command while feeding three diagnostic frames back onto the bus.

CAN exchange at a glance

Frame (name — key signals)	**Default CAN IDs (hex)**	**Dir.**	**Role inside EPS logic**
STEERING_CONTROL (torque, enable)	0x0E4	→	ADAS request, counter + chk
ENGINE_DATA (speed)	0x316	→	Assist curve vs. vehicle speed
VSA_STATUS (brake, ESP flags)	0x18F	→	Road-wheel slip & brake bias
WHEEL_SPEEDS (FL … RR)	0x3E4	→	Fine-grained speed feedback
STEER_MOTOR_TORQUE (value, flags)	0x1AB	←	10-bit torque, 2-bit ctr, 4-bit chk
STEERING_SENSORS (angle, rate)	0x17E	←	Raw shaft angle / rate for EON, HUD
STEER_STATUS (torque sens., active)	0x18A	←	Motor load + "control-active" bit

Table 4.6: Key CAN frames processed by the EPS ECU (Dir. → = consumed, ← = produced).

Processing pipeline (per tick).

1. *Driver model* selects a new target wheel-angle, easing-in/out over 0.5 s to 3 s; a small jitter is injected when centred to mimic micro-corrections.

2. **Assist computation**

   - base torque = $\theta/45^\circ$ scaled by a speed-sensitive factor;
   - if an ADAS request is *present and recent* (<500 ms) it overrides the driver torque after mapping $\pm 3840\,\text{LSB} \rightarrow \pm 4\,\text{Nm}$;
   - wind, gradient and extreme temperatures modulate the result by at most ±20%.

3. Torque is ramp-limited to $\pm 3\,\text{Nm} \cdot \text{s}^{-1}$ to emulate motor inertia and protect downstream physics.

4. **Signal generation**

   - `STEER_MOTOR_TORQUE` — 10-bit unsigned value + 2-bit counter + Honda 4-bit checksum;
   - `STEERING_SENSORS` — signed angle (-0.1°/LSB) and rate (-1°/s LSB) with identical counter/cksum;
   - `STEER_STATUS` — low-rate health report containing motor torque sensor and the "control active" flag.

The resulting behaviour feels authentic to the driver model yet leaves ample room for cyber-experiments: injecting bogus torque on `STEERING_CONTROL` instantly forces the wheels, while forged checksums or counter jumps trigger EPS debug logs — useful hooks for intrusion-detection research.

4.3.6 Gas-Pedal "Interceptor"

In many aftermarket retro-fits the original throttle-position sensor is left in place but its signal is *intercepted*, filtered and re-injected onto the CAN bus. The **Interceptor ECU** in CANvas mimics that behaviour: it listens for the high-level `GAS_COMMAND` generated by the "autopilot" (EON) and re-encodes it as a dual-channel `GAS_SENSOR` frame expected by the Power-train Controller (PCM).

CAN exchange

Frame (name — key signals)	Default CAN IDs (hex)	Dir.	Purpose inside Interceptor
GAS_COMMAND	0x1A1	→	Desired pedal percentage from EON (6 B)
GAS_SENSOR	0x1A0	←	Redundant ADC replica for PCM (6 B)

Table 4.7: Key CAN frames processed by the INTERCEPTOR ECU (Dir. → = consumed, ← = produced).

Processing steps every 100 ms.

1. **Decode** the two 16-bit command channels $x_{1,2} := \texttt{raw}/\text{scale} - 83.3$ ($\texttt{scale}_1$=0.254, $\texttt{scale}_2$=0.127) and average them to obtain a normalised gas request $g \in [0, 1]$.

2. Verify the enable flag, 4-bit rolling counter and an 8-bit additive checksum; stale frames (>1 s) raise `FAULT_TIMEOUT`.

3. Apply environment multipliers $g \leftarrow g \underbrace{f_{\text{alt}}\, f_{\text{grade}}\, f_{\text{temp}}}_{\leq 1.5}$ where altitude can derate throttle to 70% at 4 km, steep grades boost it, and extreme temperatures cut it by 20%.

4. Re-encode the final value into `GAS_SENSOR`: two identical raw channels, bit-7 = *enabled*, bits 4-6 = *fault state*, bits 0-3 = pass-through counter; no checksum per Honda DBC.

5. Expose internal diagnostics via the shared `interceptor_state` (current gas, fault enum, enable).

With this design researchers can:

- toggle enable/disable bits to see how the PCM reacts to a vanishing pedal;
- replay `GAS_COMMAND` frames with incorrect counters to trigger fault flags;
- measure how altitude or gradient manipulation propagates into longitudinal vehicle dynamics through the PCM loop.

4.3.7 Steering-Column Module (SCM)

The **SCM** emulates the electronics found inside a modern steering column: indicator stalks, wiper lever, head-light rotary switch and the cruise-control buttons on the wheel. Its thread ticks every **20 ms** so that "human-paced" inputs feel instantly responsive throughout the simulator.

CAN exchange

Frame (name — key signals)	Default CAN IDs (hex)	Dir.	Role inside SCM
ENGINE_DATA (speed)	0x316	→	Actual vehicle speed for blinker timing and auto-cancel
STEERING_SENSORS (angle, rate)	0x18A	→	Wheel angle triggers auto-cancel and lane-change logic
SCM_BUTTONS (cruise bits, lights sel.)	0x1A2	←	8-B master command frame (Honda checksum + counter)
SCM_FEEDBACK (blinkers, wiper speed)	0x294	←	Mirrors status lamps for cluster / ADAS
STALK_STATUS (auto-lights, HB hold)	0x374	←	Extra stalk states (FD CAN, 8 B)
STALK_STATUS_2 (low-beam, DRL, wipers)	0x37B	←	Second frame with remaining bits

Table 4.8: Key CAN frames processed by the SCM ECU (Dir. → = consumed, ← = produced).

Internal logic – one tick overview

1. **Blinker finite-state-machine** four states (*OFF, ON, CANCELLING, LANE_ CHANGE*) decide both lamp and CAN fields. *Auto-cancel*: wheel angle returns within ±10°; *Lane-change*: < 600 ms tap gives exactly three flashes.

2. **Wiper speed** chosen from {off, low, mid, high} as a function of `environment.weather` and `traffic_density`.

3. **Auto-head-light** controller (time-of-day, weather, ambient lux) DRL / low-beam / high-beam, including automatic high-beam hold when traffic is sparse.

4. **Cruise-control button generator** every 120 s a pseudo-random button (SET, RESUME, CANCEL, ACCEL, DECEL) is injected together with a realistic target speed (30 km/h to 120 km/h); the logic is shared with the PCM section through `CRUISE_DATA`.

5. Pack the three CAN frames (`SCM_BUTTONS`, `SCM_FEEDBACK`, `STALK_STATUS & _2`) attaching a 4-bit rolling counter and the Honda 4-bit additive checksum ($\sum$ data $+ ID +$ len) mod 16.

The SCM therefore drives all driver-facing switches while remaining fully deterministic and replayable, which is vital for cyber-physical experiments: attackers can spoof wiper or head-light messages, the IDS can correlate them with the ground-truth states kept inside the simulation, and learners can immediately spot conflicts on the virtual dashboard.

4.3.8 Vehicle-Stability Assist (VSA)

The VSA controller closes the chassis-dynamics loop at a brisk 20 ms so that wheel-slip and yaw-rate corrections feel instantaneous.

CAN exchange

Frame (name — key signals)	Default CAN IDs (hex)	Dir.	Use in VSA logic
GAS_SENSOR (redundant ADCs)	0x1A0	→	Throttle percentage for longitudinal load-transfer estimate
BRAKE_COMMAND (ADAS/AEB)	0x1FA	→	Computer-brake demand, detects AEB engagement
STEERING_SENSORS (angle, rate)	0x18A	→	Yaw target and lateral accel feed-forward
STEER_STATUS (EPS fault bits)	0x18F	→	Adds or clears ESP error codes
ENGINE_DATA (speed, RPM)	0x316	→	Reference vehicle speed for slip detection
VEHICLE_DYNAMICS (yaw, lat-accel)	0x3D2	→	Optional IMU for high-fidelity testing
WHEEL_SPEEDS (FL/FR/RL/RR) ABS	0x3E4	←	Per-wheel speeds (0.01 km/h LSB) for cluster
VSA_STATUS (brake, ESP flags)	0x18F	←	Brake pressure, wheels-moving bits, ESP / TCS state

Table 4.9: Key CAN frames processed by the VSA ECU (Dir. → = consumed, ← = produced).

Internal cycle

1. **Mirror driver inputs** – throttle from `GAS_SENSOR`, brake demand from ADAS.

2. **Wheel-speed synthesis** vehicle speed × road-factor (0.8 wet, 0.5 ice) + small random variation → four individual wheel speeds.

3. **Acceleration estimation** $a_{\text{lat}} = \theta_{\text{steer}}\, v/100 + w_{\text{wind}} k$; $a_{\text{long}} = (\text{throttle} - \text{brake})/100$.

4. **ESP / TCS heuristics** sets `esp_status` bits when *icy* road → ESP ON, *wet* road → traction control.

5. **Controlled-standstill flag** when wheels $< 1\,\text{km}\,\text{h}^{-1}$ & brake > 0.

6. Pack **WHEEL_SPEEDS** (8 B) and **VSA_STATUS** (4 B), append 2-bit rolling counter + Honda 4-bit checksum.

This minimal yet self-consistent dynamics model lets other ECUs (EPS, ADAS, cluster) experience believable yaw-rate and traction behaviour while still accepting arbitrary message injections — a prerequisite for repeatable security experiments on cyber-physical attacks.

4.4 Vehicle-Dynamics Core

The dynamics engine is integrated into the PCM (Powertrain Control Module) thread and calculates physically-plausible motion at a 100 ms update rate. It primarily focuses on longitudinal dynamics, with the following components.

4.4.1 Longitudinal Dynamics

To simulate vehicle acceleration behavior, we use a simplified longitudinal model that captures the main forces acting on the vehicle and adapted to our simulation needs [40] .

At every 100 ms tick, the engine calculates acceleration as:

$$a = \underbrace{a_{\max} \cdot \theta}_{\text{propulsion}} - \underbrace{a_{\max} \cdot b}_{\text{braking}} - \underbrace{\tfrac{1}{2}\rho C_d A v^2/m}_{\text{aero drag}} - \underbrace{C_{rr} g}_{\text{rolling}} - \underbrace{g \sin\gamma}_{\text{slope}}$$

where:

- θ is the throttle position (0–1)
- b is the brake force (0–1)
- $a_{\max}$ is the maximum acceleration/deceleration
- γ is the road gradient from the environment parameters

Speed is then updated using Euler integration:

$$v_{t+\Delta t} = v_t + a \cdot \Delta t$$

This model enables the vehicle to slow on slopes, coast on flat surfaces, and react to throttle and brake inputs. It also accounts for aerodynamic resistance and rolling friction.

4.4.2 Engine and Transmission Model

The PCM thread also simulates how the engine and gearbox respond to driver inputs. The model includes:

- Torque generation based on engine speed and throttle position
- Reduced power when the engine gets too hot
- Automatic gear shifting based on vehicle speed
- Cruise control logic to hold a steady speed

To simulate engine temperature changes, we include both heat from engine operation and cooling from air flow:

$$\dot{T}_e = \underbrace{100 \cdot \frac{\text{RPM}}{6000}}_{\text{heat input}} - \underbrace{0.1 \cdot (1 + \frac{v}{1000}) \cdot (T_e - T_{amb})}_{\text{cooling rate}}$$

where v is the vehicle speed (in km/h) and T_{amb} is the ambient temperature.

This engine model helps simulate realistic behavior such as overheating or load compensation, and ensures other ECUs can respond in real time to vehicle state changes.

4.5 Sensor Realism & Timing Jitter

Real vehicles produce noisy, imperfect signals, and attackers often exploit small timing or signal inconsistencies. To better reflect this, CANvas adds three layers of realism before transmitting any CAN frame:

1. Analog sensor noise White Gaussian noise is applied to raw sensor values, with component-specific levels. For example, wheel speed has a standard deviation of $0.05\,\mathrm{km\,h^{-1}}$, and steering angle varies by up to 0.02°.

2. Digital signal filtering Real ECUs usually smooth data before sending it. We simulate this by applying a first-order IIR filter ($\alpha = 0.2$) to continuous signals like speed or RPM. On/off signals like door switches or light stalks are left unchanged.

3. Timing jitter To simulate real-world clock drift, each CAN frame's transmission time is randomly shifted by a uniform value in $\mathcal{U}(-5, +5)$ ms. This reflects the oscillator jitter typical in low-cost automotive microcontrollers.

This provides a more authentic environment for training and evaluating timing-sensitive IDS systems.

Chapter Summary

This chapter thoroughly detailed CANvas' technical implementation, providing insights into DBC parsing, ECU logic, simulation realism and environmental factor integration. In the next chapter, we explore the implementation of the attack modules showing how each is structured, loaded, and executed within the framework and introduce the Vehicle State Interface, which lets users monitor in real time how their attacks impact vehicle behavior and individual ECUs. Through live dashboards of speed, torque, sensor readings, and fault flags, researchers gain fine-grained visibility into system responses as their scenarios unfold.

Chapter 5
Attack Surface in CANvas

Building on the implementation presented in Chapter 4, this chapter analyzes the attack surface modeled within CANvas and demonstrates how the platform supports controlled and reproducible cybersecurity experiments. We begin by outlining the general principles behind in-vehicle network attacks, then describe CANvas's capabilities as an experimentation environment, and finally introduce the modular attack scripts.

5.1 Introduction to Automotive Cyber-Security Attacks

Modern vehicles comprise dozens of ECUs that exchange thousands of CAN frames per second. While this frequent communications enables advanced driver assistance, it also exposes a rich attack surface: malicious actors can inject spoofed frames, replay stale traffic, tamper with counters or checksums, and manipulate inter-frame timing to cause unsafe or unexpected vehicle behavior.

To structure our threat analysis we draw on the *Behavioural Threat Model* (BTM) articulated for automotive networks in [37]. Inspired by that work, we use BTM as a recipe to:

- enumerate *adversary capabilities* (e.g. raw frame injection, ECU impersonation, bus arbitration);
- map those capabilities to concrete *attack objectives* such as unauthorised acceleration, phantom braking, or driver-display deception; and
- craft exploits that remain plausible within each ECU's control logic and safety envelopes.

This BTM-driven methodology ensures that every attack module in CANvas is both technically feasible and behaviourally consistent with real-world automotive constraints.

5.2 CANvas as a Cybersecurity Experimentation Platform

CANvas is designed to provide a practical, modular environment for studying automotive cybersecurity in a controlled simulation setting:

- **Representative ECU simulation:** Each ECU models realistic timing, message structures, and control behavior inspired by production systems.
- **Live injection observability:** Attack modules can inject or alter CAN frames during runtime, while internal state changes are tracked via the dashboard interface.
- **Flexible scenario design:** Researchers can configure target CAN IDs, injection timing, attack frequency, and simulated environmental conditions to tailor experiments.

5.2.1 Real-time Vehicle State Monitoring Interface

To observe attack impacts in real time, CANvas provides a comprehensive dashboard (Figure 5.1) that displays every ECU's key parameters alongside global vehicle and environmental states. To provide structured insight into what the dashboard displays, Table 5.1 summarizes the monitored parameters for each ECU and environmental context.

Section	Field	Description
Environment	Weather	Road weather condition
	Road Condition	Surface grip category
	Gradient (%)	Road incline/decline
	Temperature (°C)	Ambient air temperature
	Altitude (m)	Elevation above sea level
	Traffic Density	Simulated traffic level [0–1]
ADAS ECU	PCM Speed (km/h)	Speed for ACC/AEB control loops
	PCM Gas	Gas request read from the bus (PCM)
	Cruise Speed (km/h)	Adaptive cruise target
	Radar Obstructed	Lead-vehicle detection flag
	Computer Brake (%)	AEB braking command
	Brake Lights	Brake lamp status
	LKAS Problem	Lane-keeping fault flag
BDY ECU	Seatbelt Status	Driver/passenger belt latch state
	Door States (FL/FR/RL/RR)	Door open/closed flags
	Trunk Open	Trunk latch state
	Airbag Status	Airbag readiness indicator
	Brake Lights	Brake lamp mirror
SCM ECU	Left/Right Blinker	Indicator lamp status
	Wiper Speed	Current wiper setting
	Lights Setting	DRL/low/high-beam mode
	Cruise Buttons	Last cruise button event
PCM ECU	Gas Pedal (%)	Normalized throttle input
	Engine RPM	Engine rotational speed in revolutions per minute
	Trans. Speed	Vehicle speed from PCM
	Torque Est	Estimated wheel torque
	Gear Shifter	P/N/R/D selector
	Cruise Speed	Active cruise target
	Brake Request	Current brake force request
VSA ECU	Wheel Speeds (km/h)	Per-wheel velocity
	Brake Pressures	Computed brake pressures
	ESP Status	Stability control active flag
	Throttle (%)	Throttle after slip correction
EPS ECU	Motor Torque (Nm)	Steering assist torque
	Steer Angle (°)	Steering wheel angle
	Steer Rate (°/s)	Angular velocity
Interceptor ECU	Commanded Gas	Gas command from EON
	Applied Gas	Final throttle to PCM
	Enabled	Interceptor active flag
EON ECU	Gas Command	Raw gas value generated by EON logic
	EON Enabled	Indicates if EON is actively engaged

Table 5.1: Dashboard parameters and descriptions by ECU module

Figure 5.1: Real-time Vehicle ECU State Monitoring Interface

With CANvas's observability and simulation infrastructure established, we now introduce the modular attack engine. Each attack module is configurable, independently executable, and tailored to explore a specific threat scenario under controlled conditions.

5.3 Attack Modules and Configuration

To systematically evaluate the resilience of in-vehicle networks, we developed a suite of configurable Python attack modules within CANvas. Each module encapsulates a specific attack scenario, allowing researchers to flexibly target different ECUs and message types. Modules support a variety of parameters that tailor attack dynamics for realism and extensibility.

All attack modules share a unified configuration interface, enabling consistent deployment and rapid experimentation:

- **Interface:** The CAN channel (e.g., `vcan0`) used for message injection.
- **Target CAN ID(s):** Specific identifiers targeted by spoofing.
- **Frequency:** Injection rate in Hz, determining the spoofing frequency.
- **Pattern:** Injection temporal pattern (continuous, pulsed, random).
- **Intensity:** Magnitude of attack effect (e.g., throttle percentage, brake force).

This uniform approach ensures rapid prototyping of new attacks and straightforward adaptation of existing modules to various scenarios.

5.4 Detailed Attack Scenarios

The following sections provide detailed descriptions of each attack scenario implemented within CANvas, outlining objectives, targeted ECUs, technical approaches, configurable parameters, and the specific security implications arising from each attack.

5.4.1 Ghost Throttle Attack

The Ghost Throttle Attack aims to induce unauthorized acceleration by spoofing `GAS_COMMAND` messages. It targets the Powertrain Control Module (PCM) via the throttle interceptor, exploiting the trust relationship between the EON driver assistance system and the throttle actuator.

The attack employs progressive throttle ramping for both stealth and effectiveness:

- Begins moderately (50%) to prevent sudden vehicle reactions.
- Increments throttle in 10% steps, emulating natural driver acceleration.
- Caps maximum throttle at 80% to avoid triggering vehicle safety interventions.
- Maintains counter sequencing to evade basic replay detection.

5.4.2 Brake Spoof Attack

The Brake Spoof Attack manipulates braking behavior either by spoofing brake commands or falsifying Vehicle Stability Assist (VSA) status. It affects multiple critical ECUs including PCM, VSA, ADAS, and BCM, enhancing its versatility and impact.

The attack operates in dual modes:

- **Command Mode:** Direct spoofing of `BRAKE_COMMAND` frames, activating emergency braking (AEB) pumps and brake lights for realistic and impactful events.
- **VSA Mode:** Low-level manipulation of `VSA_STATUS` frames, altering perceived brake pressure and system statuses to provoke unpredictable vehicle responses.

Attack patterns include continuous, pulsed, and random activations, each introducing different hazard dynamics and complicating detection.

5.4.3 Door Status Spoof Attack

This attack falsifies door status information, impacting ECUs such as the Instrument Cluster and security/dynamic control systems. By precisely manipulating bits representing door states, attackers can create false safety alarms and influence systems reliant on door detection (e.g., vehicle speed limiters).

Attack patterns include continuous, pulse, and random behaviors, enabling diverse and potentially disruptive effects.

5.4.4 Turn Signals Spoof Attack

By falsely activating turn signals, this attack misleads drivers and disrupts ADAS systems. Targeting the Instrument Cluster and related ADAS modules, the attack offers extensive directional and temporal control options, ranging from consistent signaling to random and chaotic flashing patterns, affecting driver perception and autonomous vehicle decisions.

5.4.5 Speed Spoofing Attack

This scenario targets critical speed-dependent ECUs such as ADAS, VSA, EPS, and the Instrument Cluster. The attack manipulates perceived vehicle speed through constant, ramping, or random reporting, potentially causing inappropriate safety responses, incorrect steering assist, false collision warnings, and significant driver confusion.

5.4.6 Headlights Spoof Attack

Aimed at the Instrument Cluster and lighting systems, this attack manipulates headlight control signals to create hazardous lighting conditions. By forcing headlights into inappropriate modes (auto, low, high, park, random) and using varied temporal patterns (continuous, pulsed, random), attackers can cause severe visibility issues, distractions, and road hazards.

5.4.7 Security Implications of Attacks

The security implications of these attacks are significant, reflecting vulnerabilities that pose direct threats to vehicle safety and functionality:

- **Ghost Throttle Attack:** Unauthorized vehicle acceleration, severely compromising driver control and road safety.
- **Brake Spoof Attack:** Sudden and hazardous braking, creating road hazards and provoking unsafe emergency responses.
- **Door Status Spoof Attack:** False safety alarms, unnecessary safety interventions, and disruptions to legitimate vehicle operation.
- **Turn Signals Spoof Attack:** Confusion and misdirection for surrounding drivers, disruption of automated driving systems.
- **Speed Spoofing Attack:** Incorrect behavior from speed-dependent systems, misleading the driver and compromising vehicle stability.
- **Headlights Spoof Attack:** Dangerous lighting states that impair driver visibility and distract other road users, increasing collision risk.

5.4.8 Attack Scenarios Summary

A concise summary of each attack, its targeted ECUs, methods, and configurable parameters is provided in Table 5.2 to complement the detailed explanations provided above.

Table 5.2: Attack Scenarios Summary

Attack	**Affected ECU(s)**	**Technical Approach**	**Configurable Parameters**
Ghost Throttle	PCM	Progressive throttle ramp, stealthy replay evasion	`GAS_COMMAND_ID`
Brake Spoof	PCM, VSA, ADAS, BCM	Dual-mode (Command/VSA), brake intensity, AEB flags, temporal variations	`BRAKE_CMD_ID`, `VSA_STATUS_ID`, Brake Level, Brake Lights, AEB, Mode
Door Status Spoof	Instrument Cluster, Security Systems	Bit-level manipulation (door/trunk states), multiple temporal behaviors	`DOORS_STATUS_ID`, Trunk status option
Turn Signals Spoof	Instrument Cluster, ADAS	Directional spoofing (left-/right/both/random), diverse timing patterns	`FEEDBACK_ID`, Side (left-/right/both/random)
Speed Spoofing	ADAS, VSA, EPS, Instrument Cluster	Speed manipulation (constant/ramp/random), adjustable speed range and increment	`SPEED_ID`, Target Speed, Min/Max Speed, Step
Headlights Spoof	Instrument Cluster, Lighting Systems	Headlight mode spoofing, temporal pattern variations	`STALK_STATUS_ID`, `STALK_STATUS_2_ID`, Mode (auto/low/high/park/random)

5.5 Summary

This chapter has comprehensively described the attack modules within CANvas, clearly outlining objectives, detailed methodologies, security implications, and configurable parameters. This structured approach facilitates precise and versatile experimentation, providing valuable insights into automotive cybersecurity. In the next chapter, we conclude our findings, critically evaluate current limitations, and propose future improvements for the CANvas platform.

Chapter 6
Conclusion, Limitations, and Future Work

This chapter summarizes the key outcomes of the CANvas project, identifies current limitations in its design, and proposes future directions for development. As limitations and future opportunities are often closely linked, we present them together to provide a coherent reflection on the platform's current state and its evolution potential.

6.1 Summary of Contributions

CANvas offers a reproducible and extensible environment for automotive cyber-physical security research. Its main contributions can be summarized as follows:

- **Faithful ECU Simulation:** Eight core ECUs—Powertrain Control Module (PCM), Vehicle Stability Assist (VSA), Electric Power Steering (EPS), Advanced Driver Assistance Systems (ADAS), Body Control Module (BCM), Steering Column Module (SCM), Interceptor, and EON—are simulated with realistic timing, message structures, and control behaviors representative of production systems.

- **Modular Attack Library:** A collection of six modular Python scripts showcases representative in-vehicle attack scenarios (e.g., ghost throttle, brake spoofing, door and lighting manipulation), with a shared configuration interface enabling rapid customization and testing.

- **Real-Time Monitoring Dashboard:** An interactive GUI displays over 30 ECU-specific state variables in real time, offering immediate insight into the behavior of the system under both normal and attack conditions.

- **Data Export for Offline Analysis:** All dashboard signals and raw CAN traffic can be exported in CSV or JSON format, facilitating reproducibility, post-processing with tools like *Pandas* or MATLAB, and sharing of results across research teams.

6.2 Current Limitations

Despite its practical capabilities, the current CANvas prototype exhibits several limitations that restrict its scope and fidelity:

1. **Single-Bus Architecture:** CANvas currently supports only classical CAN. Advanced protocols such as CAN FD, FlexRay, and Automotive Ethernet are not yet implemented. Supporting these technologies would require refactoring the message processing pipeline and extending timing logic to accommodate larger payloads and different arbitration schemes.

2. **Simplified Vehicle Dynamics:** The platform models vehicle movement using first-order approximations. It does not yet simulate detailed effects such as tire slip, suspension behavior, or three-dimensional aerodynamics, which could impact the realism of experiments involving high-speed or non-linear motion.

3. **Software-Only Execution:** CANvas operates purely in a software-in-the-loop (SIL) environment, meaning it lacks physical or manual interaction through human-in-the-loop (HIL) components, which would typically involve real-time hardware interfacing and decision-making inputs.

4. **Timing and Physical-Layer Limitations:** Real-time accuracy depends on the host OS scheduler. Furthermore, low-level physical-layer behaviors—such as bit-stuffing, voltage collision handling, or bus contention—are not emulated, which limits testing at the bit or symbol level.

5. **Incomplete Security Feature Coverage:** The current implementation does not include cryptographic protections such as message authentication codes, secure boot procedures, or hardware trust anchors. These are increasingly standard in modern ECUs and represent an important area for future integration.

6.3 Future Work

Several enhancements could substantially increase the utility, realism, and applicability of CANvas. Below are three recommended directions:

1. Multi-Bus and Gateway Integration. Adding support for CAN FD and Automotive Ethernet, along with a configurable gateway ECU, would enable the simulation of cross-domain communication and intrusion detection strategies. This would better reflect the architecture of modern vehicles.

2. High-Fidelity Dynamics and Sensor Simulation. Integrating CANvas with external physics engines such as *CARLA* or *Chrono* would enable more realistic modeling of road interactions, suspension behavior, and sensor outputs (e.g., camera, LiDAR). This would support research into perception attacks and vehicle control under compromised conditions.

3. Security Feature Expansion. Incorporating cryptographic protocols into ECU communication, including MACs and key exchange mechanisms, would align CANvas with current industry practices and enable testing of secure in-vehicle communication schemes.

6.4 Concluding Remarks

CANvas bridges the gap between conceptual security research and practical experimentation. Its modular design, real-time visualization tools, and scenario-specific attack scripts provide a flexible foundation for studying cyber–physical interactions in connected vehicles. While the current prototype emphasizes realism and accessibility, addressing the identified limitations could position CANvas as an adopted open-source benchmark for automotive cybersecurity research and education. All artifacts developed during this

project—including source code, configuration files, and attack scripts—are available on GitHub:[1].

[1] `https://github.com/da4nyy/CANvas`

Bibliography

[1] Sam Curry. Hacking kia: Remotely controlling cars with just a license plate. https://samcurry.net/hacking-kia, 2024. Accessed: 2025-03-03. [Cited on pages II and 10.]

[2] Jesal Shah. Understanding can — a beginner's guide to the controller area network protocol. https://www.circuitbread.com/tutorials/understanding-can-a-beginners-guide-to-the-controller-area-network-protocol, 2024. Accessed: 2024-10-29. [Cited on pages VII and 7.]

[3] Murat Bozdal, Mohamed Samie, Sheeraz Aslam, and Ian Jennions. Evaluation of can bus security challenges. *Sensors*, 20(8):2364, 2020. Accessed: 2024-11-04. [Cited on pages VII, 1, 7, 8, and 9.]

[4] Influx Technology. Understanding CAN DBC. https://www.influxtechnology.com/post/understanding-can-dbc, 2021. Accessed: 2025-01-02. [Cited on pages VII and 27.]

[5] Vector Informatik. Industry trends 2019: Convergence drives competitiveness and innovation. Technical report, Vector Informatik GmbH, Stuttgart, Germany, 2019. [Cited on page 1.]

[6] Statista. Automotive electronics worldwide - statistics facts. https://www.statista.com/topics/7983/automotive-electronics-worldwide/. Accessed: 2025-04-22. [Cited on page 1.]

[7] Johan van der Griendt. Cantot: Modular can bus penetration testing framework. https://github.com/shipcod3/canTot, 2020. Accessed: 2025-01-21. [Cited on page 2.]

[8] Smith Craig. Icsim — instrument cluster simulator. https://github.com/zombieCraig/ICSim, 2016. Accessed: 2025-1-21. [Cited on pages 2, 13, and 14.]

[9] comma.ai. Opendbc: Community-maintained dbc files for production vehicles. https://github.com/commaai/opendbc/tree/master/opendbc/dbc, 2025. Accessed: 2025-04-22. [Cited on pages 2 and 19.]

[10] CANopen US. History of can. https://canopen.us/home/history-of-can, n.d. Originally published in *Control Engineering* and the CAN Newsletter. [Cited on page 6.]

[11] Bosch GmbH. Can specification version 2.0. http://esd.cs.ucr.edu/webres/can20.pdf, 1991. [Cited on page 6.]

[12] ISO 11898:1993—road vehicles — interchange of digital information — controller area network (can) for high-speed communication. https://www.iso.org/standard/20380.html, 1993. [Cited on pages 6 and 7.]

[13] The Morey Corporation. Obd-i to obd-ii: A history of on-board diagnostics. https://www.moreycorp.com/obd-i-obd-ii-a-history-of-on-board-diagnostics/, 2022. [Cited on page 6.]

[14] United states: On-board diagnostics (light-duty vehicles). https://www.transportpolicy.net/standard/us-on-board-diagnostics/, 2023. Mandatory CAN (ISO 15765-4) from model-year 2008; accessed 02-May-2025. [Cited on page 6.]

[15] ISO 11898-1:2003—road vehicles —controller area network (can) — part 1: Data link layer and physical signalling, 2003. [Cited on page 6.]

[16] ISO 11898-2:2003—road vehicles —controller area network (can) — part 2: High-speed medium access unit, 2003. [Cited on page 6.]

[17] ISO 11898-3:2006—road vehicles —controller area network (can) — part 3: Low-speed, fault-tolerant, medium-dependent interface, 2006. [Cited on page 6.]

[18] Wikipedia contributors. Can fd. https://en.wikipedia.org/wiki/CAN_FD, n.d. Accessed: 2024-11-09. [Cited on page 6.]

[19] ISO 11898-1:2015—road vehicles —controller area network (can) — part 1: Data link layer and physical coding sub-layer, 2015. [Cited on page 6.]

[20] ISO 11898-2:2016—road vehicles —controller area network (can) — part 2: High-speed physical layer with selective wake-up functionality, 2016. [Cited on page 6.]

[21] CiA Special Interest Group CAN XL. Can xl protocol features. https://www.can-cia.org/can-knowledge/can-xl, December 2018. Accessed: december 2, 2024. [Cited on page 6.]

[22] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462, 2010. [Cited on pages 8 and 10.]

[23] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Security Symposium*, pages 77–92. USENIX Association, 2011. Accessed: 2025-05-02. [Cited on page 8.]

[24] Luca Palanca, Andrea Marchetto, Matteo Migliardi, and Mario Baldi. A stealth, selective, link layer denial-of-service attack on can bus. In Stefania Gnesi and Tiziana Margaria, editors, *Computer Safety, Reliability, and Security (SAFECOMP 2017)*, volume 10489 of *Lecture Notes in Computer Science*, pages 235–247. Springer, 2017. Accessed: 2025-05-02. [Cited on pages 8 and 10.]

[25] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. Technical report, IOActive Labs, 2015. Black Hat USA presentation; Accessed: 2025-04-21. [Cited on pages 9 and 10.]

[26] Christiaan Beek. Defcon – connected car security. https://www.mcafee.com/blogs/other-blogs/mcafee-labs/defcon-connected-car-security/, 2018. Accessed: 2025-05-02. [Cited on pages 9 and 10.]

[27] Andy Greenberg. Hackers can steal a tesla model s in seconds by cloning its key fob. https://www.wired.com/story/hackers-steal-tesla-model-s-seconds-key-fob/, 2019. Accessed: 2025-05-02. [Cited on pages 9 and 10.]

[28] Ken Tindell. Canhack pico: A diy board for exploring can protocol attacks. https://kentindell.github.io/2021/02/06/canhack-pico/, 2021. Accessed: 2025-05-01. [Cited on page 9.]

[29] Ken Tindell. Can injection: How to steal a car using a headlamp connector. https://kentindell.github.io/2023/04/03/can-injection/, 2023. Accessed: 2025-05-02. [Cited on pages 10 and 11.]

[30] Canis Labs. CryptoCAN: A lightweight cryptographic scheme for CAN bus. https://canislabs.com/cryptocan/, 2023. Accessed: 2025-05-02. [Cited on pages 11 and 12.]

[31] Lars Völker. Comparing automotive network security for different communication technologies. https://automotive-network-security.com/papers/2018-01-31_AutomotiveEthernetCongress_DrLarsVoelker_final.pdf, 2018. Presentation at the Automotive Ethernet Congress, January 31, 2018. [Cited on page 11.]

[32] Canis Labs. Can-hg: Hardware guard for CAN protocol integrity. https://canislabs.com/canhg, 2021. [Cited on page 11.]

[33] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. Security threats to automotive can networks—practical examples and selected short-term countermeasures. *Reliability Engineering System Safety*, 96(1):11–25, 2011. Special Issue on Safecomp 2008. [Cited on page 11.]

[34] Siti-Farhana Lokman, Abu Talib Othman, and Muhammad-Husaini Abu-Bakar. Intrusion detection system for automotive controller area network (can) bus system: a review. *EURASIP Journal on Wireless Communications and Networking*, 2019(184), 2019. Accessed: June 2, 2025. [Cited on page 11.]

[35] Vector Informatik GmbH. CANalyzer – Analysis and Stimulation of CAN Networks. https://www.vector.com/int/en/products/products-a-z/software/canalyzer/, 2025. Accessed: 2025-05-02. [Cited on pages 13 and 14.]

[36] Andreas Finkenzeller, Anshu Mathur, Jan Lauinger, Mohammad Hamad, and Sebastian Steinhorst. Simutack - an attack simulation framework for connected and autonomous vehicles. In *2023 IEEE 97th Vehicular Technology Conference (VTC2023-Spring)*, pages 1–7, 2023. [Cited on pages 13 and 14.]

[37] Jo Laufenberg, Thomas Kropf, and Oliver Bringmann. A framework for can communication and attack simulation. In *2022 IEEE 95th Vehicular Technology Conference: (VTC2022-Spring)*, pages 1–7, 2022. [Cited on pages 13, 14, and 39.]

[38] CANToolz: Modular framework for CAN bus security testing. https://github.com/CANToolz/CANToolz, 2025. Accessed: 2025-05-02. [Cited on pages 13 and 14.]

[39] CarSim: Vehicle Dynamics Simulation Software. https://www.carsim.com/, 2024. Accessed: 2024-05-02. [Cited on page 18.]

[40] Rajesh Rajamani. *Vehicle Dynamics and Control*. Springer, New York, 2nd edition, 2011. [Cited on page 37.]