Securing Connected and Automated Vehicle through Proactive Vulnerability Analysis and Security Enhancement

by

Shengtuo Hu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Doctoral Committee:

Professor Z. Morley Mao, Chair
Assistant Professor Baris Kasikci
Assistant Professor Neda Masoud
Professor Atul Prakash

Shengtuo Hu
shengtuo@umich.edu
ORCID iD: 0000-0003-1687-1081

To my family and beloved.

ACKNOWLEDGMENTS

The most unique and rewarding journey in my life is finally ending. I will always miss the past five years, filled with many beautiful memories. Looking back on the past years, I feel extremely thankful for all my friends who helped me reach the highs and through the lows. Without their company, I would not have been able to overcome challenges and finish the Ph.D. journey. To quote a motto from the magazine *Nirvana Weekly* in my high school: "*Through the darkest dark, may we see the light*."

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Z. Morley Mao, who has offered continuous support and guidance at every stage of my Ph.D. study. I feel truly fortunate to be her student, who learned independent research, critical thinking, and project management under her supervision. Her expertise in computer science has inspired me to dive deeper into cybersecurity research. Her insightful comments always keep me thinking about unnoticed flaws and directions for improvement in my research. I will never forget her encouragement to aim high and not easily give up, as persistence will finally pay off.

I would also like to extend my sincere thanks and respect to Prof. Neda Masoud, Prof. Baris Kasikci, and Prof. Atul Prakash for serving on my committee. Their valuable input is very much appreciated, which helps me shape my dissertation better.

Besides, I am very grateful to my research collaborators: Prof. Qi Alfred Chen from the University of California, Irvine, Prof. Yiheng Feng from Purdue University, and Dr. André Weimerskirch from Lear Corporation/University of Michigan Transportation Research Institute. Their constructive feedback contributed immensely to this dissertation, and I really enjoyed working with them. As well, I want to thank Dr. Yaohui Chen, Dr. Cornelius Aschermann, Hasnain Lakhani, and Dr. Yuanqi Shen for their kind support during the internship at Meta. They showed their appreciation

for my work during the internship. I learned a great deal from them about engineering skills and career growth, which will still benefit me in the future. I would also like to acknowledge the support from the National Science Foundation and Mcity at the University of Michigan.

I would like to give my special thank to Prof. Daniel Xiapu Luo from the Hong Kong Polytechnic University. His guidance in the early stage of my research path broadened my horizon in cybersecurity research. His advice on both research and career path will always benefit me.

I would like to express my appreciation to my labmates from the RobustNet Research Group: Prof. Qi Alfred Chen, Dr. Yihua Guo, Dr. Yunhan Jia, Dr. Yuru Shao, Dr. Shichang Xu, Dr. Ke Hong, Dr. Yikai Lin, Dr. Xiao Zhu, Yulong Cao, Jiachen Sun, Xumiao Zhang, Won Park, Jiwon Joung, Can Carlak, Qingzhao Zhang, Shuowei Jin, Ruiyang Zhu, Wenyuan Ma, Minkyoung Cho, and Xueshen Liu. I feel honored to work with these talented people and wish all of them the best of luck in their future life. To my friends at Ann Arbor, Dr. Zihao Deng, Wenyi Liu, Dr. Jingcheng Xiao, Dr. Ruiting Li, Dr. Wenhao Shao, Xinjing Huang, Sicen Du, Guanglong Huang, Hongling Lu, etc., I am fortune to meet you all. I will always remember the happy moments between us.

Last but not least, I would like to thank my family and my partner. To my grandparents Duanlin Hu, Aier Wei, Daoyou Song, Peigui Li, my parents Yubin Hu, and Bifang Song, I cannot thank them enough for their support and sacrifice over the past decades. Words could hardly express how much my family means to me. They always stand with me, care about me, and love me unconditionally, no matter what happens. I miss my deceased grandpas, Duanlin Hu and Daoyou Song, and will miss them forever. I still feel sad for not being able to stay with them in their last moments. I hope they are resting in peace. To my partner, Muru Zhou, I feel incredibly fortunate to have met you at the beginning of our Ph.D. journeys. Since then, it has been hard for me to imagine life without you. Thank you for accepting me, supporting me, and sharing your pain and joy with me. Your emotional support is essential to me, particularly during the pandemic, which helped me through the lows. I hope we will still chase our dreams together, and always believe you will eventually make your art dream come true.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

ABSTRACT

The rapidly evolving Connected and Autonomous Vehicle (CAV) technology brings new security challenges to vehicular systems, because newly introduced communication and system components inevitably increase the attack surface of vehicles if being abused, leading to potential safety hazards on the road. For example, the emerging Connected Vehicle (CV) technology, which enables vehicles to exchange safety and mobility information wirelessly (e.g., location and speed) with traffic infrastructure and other vehicles, opens a door for spoofing attacks. On the other hand, the development of Autonomous Vehicle (AV) results in the increasing data transfer needs of various sensors (e.g., cameras, LiDAR), which stimulates the adoption of Automotive Ethernet, the next-generation in-vehicle network. However, no common standard has been established for the security protocol of the in-vehicle Ethernet network. Therefore, it is highly desirable to systematically understand vulnerabilities in the current CAV systems and the corresponding security/safety consequences to proactively uncover and address these flaws before large-scale deployment.

To achieve this goal, in this dissertation, we demonstrate that rigorous techniques, such as formal methods, program analysis, and the trusted execution environment (TEE), can be used for proactive vulnerability discovery and security enhancement in the safety-critical CAV system. At the design level, we leverage formal methods to uncover design flaws and ensure the security guarantee of the proposed defense. To study the emerging CV network interface, we propose a model-checking-based approach, CVANALYZER, that harnesses the attack discovery capability of the general model checker and the quantitative threat assessment of the probabilistic model checker to automate the analysis. For in-vehicle Ethernet security, we present GATEKEEPER, a gateway-based source authentication protocol. Except for the source authentication property, we then verify that GATEKEEPER can defend against the spoofing attack and alleviate the impact of the DoS

attack. At the implementation level, we employ both static and dynamic program analysis. To defend against the spoofing attack, we build a TEE-based defense system, CVSHIELD, to protect the integrity of the sensor data reading and processing pipeline. To uncover semantic vulnerabilities in the CAV system, we prototype CAVFUZZER that incorporates a novel object-level mutator and utilizes the data-flow feedback to guide the fuzzing process.

CHAPTER 1

Introduction

Due to the complexity and safety-critical nature, the rapidly evolving Connected and Autonomous Vehicle (CAV) technology brings new security challenges to vehicular systems. The CAV system introduces new modules, like the Connected Vehicle (CV) communication technology, various perception sensors, and the Automotive Ethernet. The CV communication module enables vehicles to wirelessly exchange safety and mobility information in real time with traffic infrastructure and other vehicles. Besides, powerful camera and LiDAR (Light Detection and Ranging) sensors are essential for the CAV system to percept the surrounding environment (e.g., obstacles). Furthermore, to support the transmission of high-bandwidth sensor data, Automotive Ethernet [124, 100, 85, 89] is considered the next-generation in-vehicle network, because of its high bandwidth, high throughput, and low cost characteristics.

However, the newly introduced modules inevitably increase the attack surface of vehicles, which can be exploited to cause safety hazards on the road. For the CV communication module, Chen et al. [37] have shown that the attacker can cause severe congestion or increased safety risks by compromising vehicles and broadcasting falsified sensor data. Also, Hu et al. [77] demonstrate that CV communication can be blocked, which eliminates the benefits of CV communication and can further result in traffic accidents. Besides, for the perception sensors, researchers have already demonstrated that the perception sensors are vulnerable to various attacks, such as the LiDAR spoofing attack [33], physical-world camera attacks [30, 80], 3D object attacks [32]. Lastly, for the Automotive Ethernet, we observe that *source authentication* and *DoS prevention* are two miss-

ing but essential security properties. Except for the malicious intents, the attacker may also want to benefit herself from the attack for personal gains. For example, the attacker can launch attacks to stop the surrounding vehicles so that the attacker can drive across the intersection quickly. Moreover, ransomware malware can affect the CAV system as well. The attacker can exploit known vulnerabilities to lock down the CAV system to request a ransom.

The objective of my dissertation is to systematically understand flaws in the current CAV system, and the corresponding security/safety consequences so that these flaws can be proactively discovered and addressed before large-scale deployment, which is challenging in multiple dimensions:

- **Compound CAV system**. The CAV system is a compound system that consists of multiple modules, including perception, localization, control, planning, prediction, and CV communication. Such a compound system naturally introduces a large attack surface. Most importantly, the system complexity raises the bar of the security analysis. For instance, analyzing the CV network protocol can incur the state explosion problem. Besides, machine learning (ML) models are commonly used in the CAV system, while the conventional approach cannot be directly applied to ML-related modules. For example, the code coverage is ineffective while fuzzing ML-related modules.

- **Cyber-physical nature**. Due to the cyber-physical nature of the CAV system, physical constraints should be included in the vulnerability discovery, especially for the attacks from the perception sensor module. Without considering the surrounding environment and the physical constraints, it is unlikely to trigger potential vulnerabilities in the physical world. For example, while fuzzing the CAV system, mutating the sensor inputs can result in program crashes [230], but not all of them can be reproduced in the physical world, such as byte-level corruption of the sensor data. Moreover, because of the exploitability, we can also ensure that identified vulnerabilities have high risks so that developers can prioritize mitigating identified vulnerabilities. Notably, it is non-trivial to incorporate the physical constraints during the analysis of the CAV system. We need to understand the modifiable areas of

different types of sensor inputs. Also, we should clearly define the attacker's capabilities against different sensors.

- **Real-time requirements**. Defense mechanisms for the CAV system should satisfy the real-time requirements of the CAV system. Since the CAV system is safety-critical, the introduced defense mechanisms should not incur high overhead; otherwise, the safety-critical behaviors may be delayed. For instance, CV Basic Safety Messages (BSMs) must be sent every 100 ms. Besides, the transmission of in-vehicle control data (e.g., in-vehicle CAN frames) has strict latency requirements ($\leq$ 10 ms). Therefore, we should design lightweight defense mechanisms and optimize the overall performance accordingly to ensure the timely reactions of the CAV system.

My research is dedicated to addressing these challenges. The overall goal is to advance the security of the CAV system through proactive vulnerability discovery and security enhancement. Specifically, utilizing formal methods, program analysis, and the trusted execution environment (TEE), this dissertation research systematically (1) detects design-level flaws in CV communication protocols, (2) designs practical broadcast authentication approach for the next-generation in-vehicle Ethernet network, (3) presents rigorous security enhancement against the CV spoofing attack, and (4) uncovers semantic vulnerabilities against the CAV system. In summary, this dissertation demonstrates that: Proactive vulnerability discovery and security enhancement of the CAV system can (1) uncover new security vulnerabilities, (2) systematically examine fundamental vulnerability causes and security consequences, and (3) provide strong security guarantees for the defense mechanisms.

In this dissertation, formal methods and program analysis are used for design- and implementation-level analysis, respectively. These two types of techniques enable automated analysis and can save human efforts in the future. Table 1.1 summarizes the techniques used in the dissertation for different analysis tasks. Specifically, for design-level analysis, model checking and formal verification (i.e., Tamarin Prover [142]) offer soundness and strong security guarantees in vulnerability discovery and security enhancement tasks. On the other hand, for implementation-

level analysis, dynamic program analysis can help us uncover exploitable vulnerabilities in complex CAV systems, and static program analysis can assist in automatically extracting sensitive code sections. By combining these techniques, we conduct systematic security analysis at the design and implementation levels for vulnerability discovery and security enhancement.

Analysis Level (Main Technique)	Analysis Task	
	Vulnerability Discovery	Security Enhancement
Design-Level (Formal method)	Model checking	Formal verification
Implementation-Level (Program analysis)	Dynamic program analysis (fuzzing)	Static program analysis + TrustZone

Table 1.1: Overview of techniques used in the dissertation.

However, due to the system complexity, applying these techniques in the CAV system is challenging. To enable the analysis, we present the following approaches for four tasks in the dissertation. First, we propose an abstraction approach, which can largely reduce the state space for design-level vulnerability discovery when protocol packet data is included in the analysis. Second, for design-level security enhancement, we follow prior works to transform DoS-related properties so that they can be verified. Third, we design a novel object-level mutator for CAV sensor data for implementation-level vulnerability analysis, which can help uncover semantic vulnerabilities for CAV systems. Last, for implementation-level security enhancement, we abstract the program that needs protections into a general representation, such that the static program analysis can automatically extract the sensitive code sections.

1.1 Overview

This dissertation validates the thesis statement stated in Chapter 1 on four technical tasks, focusing on different attack surfaces.

1.1.1 CV Network

We perform the first rigorous security analysis to automate the discovery of availability or DoS (Denial of Service) vulnerabilities, in (1) the latest version of the IEEE 1609 protocol family and (2) Cooperative Adaptive Cruise Control (CACC) applications (i.e., platoon management protocols (PMPs)). To achieve the analysis goal, we design a novel system, CVANALYZER, that leverages (1) a general model checker (MC) [231] and (2) a probabilistic model checker (PMC) [111] to automate *both* the attack discovery and the attack assessment. CVANALYZER successfully uncovers 4 *new* DoS vulnerabilities in Peer-to-Peer Certificate Distribution (P2PCD), which can prevent the application layer from processing incoming packets, and 15 vulnerabilities (14 of 15 are new) in PMPs, which can block the communication among platoon members. For these newly-discovered vulnerabilities, we have constructed practical exploits and validated them in a real-world testbed. We have also reported to and received confirmations for P2PCD attacks from IEEE 1609 Working Group [94]. Besides, our case studies demonstrate that P2PCD attacks can lead to traffic accidents, and PMP attacks can affect the speed stability of the victim vehicle. Lastly, we discuss the fundamental reasons for each identified vulnerability and propose effective mitigation solutions.

1.1.2 In-Vehicle Network

We conduct a systematic analysis of MACsec, IPsec, and TLS for the next-generation in-vehicle Ethernet network, covering security and performance requirements, which shows that *source authentication* and *DoS prevention* are two missing but essential security properties for these candidates. To address identified limitations, we propose a novel gateway-based broadcast authentication protocol, GATEKEEPER, to ensure source authentication for the in-vehicle Ethernet network. Additionally, we integrate a DoS protection approach, based on the time-lock puzzle [185], to alleviate the impact of an aggressive attacker who aims at frequently triggering computationally heavy operations at the authenticator. Moreover, we formally verify that GATEKEEPER achieves the desired security properties, using the Tamarin prover [142], which further strengthens the security guarantee of GATEKEEPER.

Although this task targets existing threats, it aims at improving the security of the next-generation in-vehicle Ethernet network. The in-vehicle security is crucial for the CAV system to ensure the correctness of processed data and timely delivery of issued control commands. Specifically, the broadcast authentication protocol, GATEKEEPER, can help users defend against the spoofing attack and alleviate the DoS threat. These two threats are common for the in-vehicle network, leading to falsified data or in-vehicle network delay. Such negative implications can further affect the functionality of the CAV system, as the CAV system eventually needs to consume and produce data in the in-vehicle network.

1.1.3 CAV System

To prevent compromised vehicles from sending falsified sensor data, we propose a system CVSHIELD, utilizing the recent advances in hardware-assisted security (e.g., ARM TrustZone). CVSHIELD can ensure the sensor data integrity from their reading to their transmission at the vehicle side. In general, we relocate all codes that are related to sensor data reading, processing, encapsulation, and transmission from the rich execution environment (REE) into the trusted execution environment (TEE). However, manually extracting code sections is laborious and error-prone. Also, we should minimize the size of the trusted computing base (TCB) in TEE to reduce the attack surface. To achieve these goals, we propose leveraging program slicing to extract code sections automatically and eliminate irrelevant codes. We demonstrate that CVSHIELD can support GPS data reading, and our optimization can eliminate the time overhead introduced by context switches of TrustZone.

1.1.4 CAV Sensors

The rapidly evolving Connected and Autonomous Vehicle (CAV) technology brings new security challenges to vehicular systems, because newly introduced communication and system modules inevitably increase the attack surface of vehicles. However, the security of the CAV system itself is largely under explored [230, 201], especially under physical attacks. The CAV system introduces

various modules and lots of library dependencies, which raise the bar of the security analysis. Notably, compromising the CAV system may jeopardize its normal functioning, leading to unexpected driving behaviors. Thus, we should thoroughly analyze the CAV system itself. For this task, we utilize fuzzing to discover potential semantic vulnerabilities in the CAV system under the physical-world attacks [30, 32, 151, 138]. We prototype CAVFUZZER, which introduces a novel object-level mutator for camera inputs and loss-based feedback to guide the fuzzer better.

1.2 Thesis Organization

This dissertation is structured as follows. Chapter 2 provides sufficient background on CV technology, in-vehicle security protocols, the trusted execution environment, and the CAV system. Related works are discussed in Chapter 2 as well. In Chapter 3, we perform the first rigorous security analysis to automate the discovery of DoS vulnerabilities in CV communication protocols. In Chapter 4, we systematically analyze the security protocols in the next-generation in-vehicle Ethernet network. In Chapter 5, we combine the trusted execution environment with the static program analysis to protect the sensor data integrity. In Chapter 6, we integrate data-flow feedback with object-level mutation strategies to uncover semantic inconsistencies in the CAV system. At last, we discuss future works and conclude the dissertation in Chapter 7.

CHAPTER 2

Background and Related Work

In this chapter, we introduce necessary technical background about the CV network (§ 2.1), in-vehicle security protocols (§ 2.2), the trusted execution environment (§ 2.3), and the CAV system (§ 2.4). After that, we present related works in these domains.

2.1 CV Technology

CV network, based on Dedicated Short Range Communications (DSRC), provides connectivity in support of mobile and stationary CV applications, which offers users (e.g., drivers) greater situational awareness of events, potential threats, and imminent hazards, intending to enhance the safety, mobility, and convenience of everyday transportation [95]. The Basic Safety Message (BSM) defined in SAE J2735 [46] is used by a variety of applications, such as Forward Collision Warning (FCW), Cooperative Adaptive Cruise Control (CACC), to exchange safety data regarding vehicle state (e.g., location and speed). The transmission rate of BSM is typically 10 times per second [68].

In the CV network, there are two basic types of devices: (1) On-Board Unit (OBU) in a roaming vehicle and (2) stationary Road-Side Unit (RSU) along the road. Usually, these devices are ARM embedded devices and install Linux operating systems [45], and the communication pattern of the CV network is individual messages that are broadcast without response [93]. As shown in Figure 2.1, the OBU is mounted in a roaming vehicle and connected with in-vehicle sensors like GPS and the in-vehicle network such as Controller Area Network (CAN). The OBU is mounted in

a roaming vehicle and connected with in-vehicle sensors like GPS and the in-vehicle network such as Controller Area Network (CAN). The RSU is a stationary unit along the road and connected with larger infrastructures or core networks such as the Internet. Roaming vehicles with OBUs installed can not only directly communicate with each other (i.e., V2V) but also communicate with RSUs (i.e., V2I), collectively called Vehicle-to-Everything (V2X) communication.

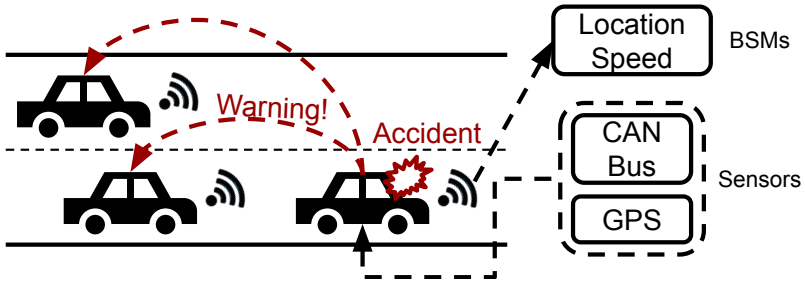


Figure 2.1: OBUs on vehicles read sensor data from GPS and CAN bus, which will be encapsulated in BSMs for transmission.

2.1.1 CV Network Stack

As shown in Figure 2.2, IEEE 802.11p [96], IEEE 1609 protocol family [91, 93, 92], and SAE J2735 [46] form the current CV network stack.

IEEE 802.11p [96] and its extension IEEE 1609.4 [92] together define the basis of the CV network stack, in which IEEE 802.11p disables the authentication, association, and data confidentiality services at the MAC layer to minimize the message latency. Above them, IEEE 1609.3 [93] defines the WAVE Short Message Protocol (WSMP), which is optimized to minimize communication overhead. The Basic Safety Message (BSM, a.k.a., the beacon message) defined in SAE J2735 is used by a variety of applications, such as Forward Collision Warning (FCW), Cooperative Adaptive Cruise Control (CACC), to exchange safety data regarding vehicle state (e.g., location and speed). The transmission rate of BSM is typically set to 10 times per second [68, 3, 4].

Due to the safety-critical nature of CV applications, IEEE 1609.2 [91] specifies security mechanisms to provide confidentiality, authenticity, integrity, and non-repudiation. It introduces digital certificates to enable digital signature (ECDSA), with the support of a Public-Key Infrastructure

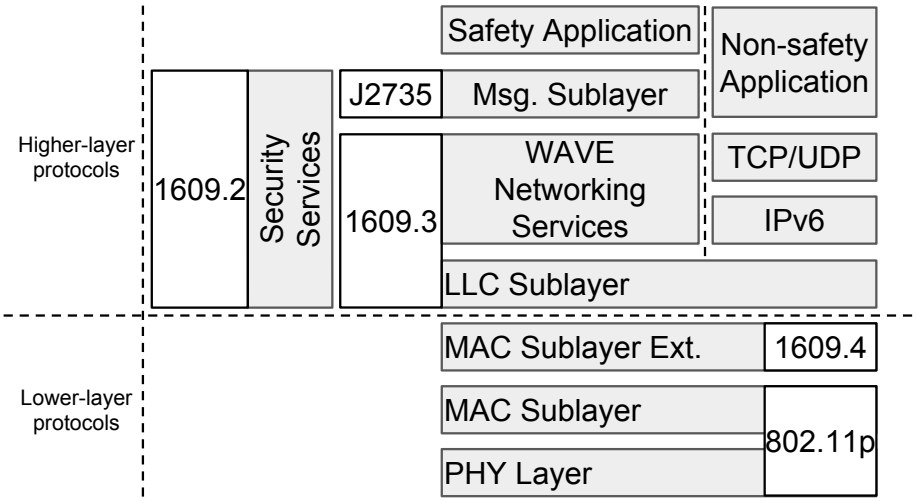


Figure 2.2: CV network stack.

(PKI) system called Security Credential Management System (SCMS) [29]. Also, SCMS supports misbehavior detection and certificate revocation to prevent malicious vehicles from communicating with others, while the development of the misbehavior detection algorithms is still ongoing.

In particular, IEEE 1609.2 specifies a unique feature called Peer-to-Peer Certificate Distribution (P2PCD) that helps a CV device to learn unknown certificates. When a device receives a signed secured protocol data unit (SPDU), it will construct a certificate chain for the signing certificate within the SPDU. The certificate chain links the signing certificate to a known *trust anchor*, which usually refers to the root certificates shared by all CV devices, so the incoming SPDU can be trusted by the receiver. However, the CV device may be unable to construct such a certificate chain due to not recognizing the issuer of the signing certificate. In this case, the received SPDU is referred to as a trigger SPDU, and the CV device will attach the P2PCD learning request field in the next outgoing SPDU to request peer devices to provide the necessary certificates to complete the chain. P2PCD learning responses, containing requested certificates, will be sent back through WSMP by peer devices. Note that, a P2PCD learning response is sent as a protocol data unit (PDU) rather than an SPDU. That is, the P2PCD learning response itself does not carry the digital signature. The current IEEE 1609.2 does not mention the verification for the payload of the learning response (cf. IEEE 1609.2-2016, Clause 8.2.4.1 c)). Besides, the P2PCD example in IEEE 1609.2 (cf. 1609.2-

2016 Clause D.4.3.6) only considers `VerifyCertificate` primitive as an optional step before `AddCertificate` primitive.

2.1.2 Platoon Management Protocol (PMP)

CVs form a platoon with minimal following distances to improve traffic density and fuel economy. The PMP is an essential component for platoon applications to control platoon maneuvers. Typically, vehicles in a platoon exchange speed, location, platoon ID, and platoon depth by broadcasting beacon messages periodically. The platoon leader has a depth of 0, increasing as we go farther. The leader acts as the coordinator and controls platoon decisions such as join/merge, split, leave, and dissolve. In this paper, We study two PMPs; since PLEXE [174] only specifies the join-at-tail maneuver that is the same as Join/Merge maneuver in VENTOS, we thus mainly follow the description of PMP in VENTOS [217].

Join/Merge Maneuver Two platoons, traveling in the same lane, can initiate a merge maneuver to form a bigger platoon. The leader of the rear platoon will send a `MERGE_REQ` to the front platoon leader, if it observes that the combined platoon size is no greater than the optimal platoon size by inspecting the beacon message from the front vehicles. Upon receiving a `MERGE_ACCEPT` from the front leader, the rear platoon leader will speed up to reduce the front spacing. Then, the rear leader sends `CHANGE_PL` to notify its followers to change the platoon leader to the front leader. Meanwhile, the rear leader switches to the follower role after sending a `MERGE_DONE` to the front platoon leader.

Split Maneuver To break the platoon into two smaller platoons, a platoon leader can either actively initiate this maneuver at a specific position, or passively trigger this maneuver when the platoon size exceeds the optimal platoon size. A platoon leader first sends a `SPLIT_REQ` to the splitting vehicle where the split should occur. After receiving a `SPLIT_ACCEPT`, the platoon leader sends a `CHANGE_PL` to make the splitting vehicle a potential leader. Besides, the platoon leader needs to inform followers behind the splitting vehicle, if any, to change their leader to the splitting vehicle. After that, the platoon leader sends a `SPLIT_DONE` to the splitting vehicle, which

then switches to the leader role.

Leave Maneuver A platoon member may initiate a leave maneuver, when approaching the destination. For the leader leave, the leader will send a `VOTE_LEADER` to all followers to vote on the new platoon leader. The newly elected platoon leader needs to send a `ELECTED_LEADER` to the current leader. Then, the leader splits at the position of the elected leader by initiating the *split maneuver*, and thus hands over the leadership to the elected leader. For the follower leave, the follower will send a `LEAVE_REQ` to the leader and wait for a `LEAVE_ACCEPT`. The leader needs to split at both the succeeding vehicle, if any, of the follower, and the follower to make it a free agent, defined as a one-vehicle platoon. At this time, the follower can slow down. Once there exists enough space for the follower to change the lane, it will send a `GAP_CREATED` to the old leader and finally leave the platoon.

Dissolve Maneuver This maneuver is only initiated by the platoon leader, who broadcasts a `DISSOLVE` to all followers. Upon receiving all `ACK` messages, all platoon members act as free agents and are free to leave.

2.2 Security Protocols for In-Vehicle Ethernet

2.2.1 MACsec

Media Access Control Security (MACsec) protocol, specified in IEEE 802.1AE standard [86], is used for securing link-layer communication in Ethernet LANs. More specifically, MACsec can provide connectionless data integrity, confidentiality, data origin authenticity, replay protection, and bounded receive delay. For each outgoing Ethernet frame, MACsec adds a security tag to the frame header and appends an Integrity Check Value (ICV) to the end of the optionally encrypted frame. In addition, IEEE 802.1X provides the MACsec Key Agreement Protocol (MKA), which discovers mutually authenticated MACsec peers and elects one as a Key Server that distributes the symmetric Secure Association Keys (SAKs) used by MACsec to protect frames.

2.2.2 IPsec

Internet Protocol Security (IPsec) is a secure network protocol suite that offers secure encrypted communication between two hosts over an Internet Protocol network. Transport and tunnel mode can be used with Authentication Header (AH) [103] or Encapsulating Security Payload (ESP) [104] protocol, which can provide similar security properties as MACsec, such as connectionless data integrity, confidentiality, data origin authenticity, replay protection, etc. Internet Key Exchange (IKE) version 2 [102], IKEv2, is a recommended key exchange protocol for IPsec.

2.2.3 TLS/DTLS

Transport Layer Security (TLS) [183] and Datagram Transport Layer Security (DTLS) [184] operate between the application layer and the transport layer. They are widely used on the Internet for securing data between communicating applications. There are two primary components in TLS/DTLS: (1) handshake protocol; (2) record protocol. Unlike TLS, which operates over a reliable transport channel, typically TCP, DTLS makes changes to TLS to accommodate applications running over datagram transport. Compared with TLS, DTLS is expected to be used more for in-vehicle communication due to the delay-sensitive nature and real-time requirements of in-vehicle traffic.

2.2.4 TESLA

As introduce in RFC 4082 [167], Timed Efficient Stream Loss-Tolerant Authentication (TESLA) [169, 168, 170] protocol aims at offering *source authentication* (a.k.a., data origin authentication [19]) for multicast/broadcast communications, where a single packet can reach millions of receivers. To ensure source authentication in an efficient way, TESLA uses symmetric cryptography with time-delayed key disclosure. At first, TESLA generates a one-way key chain for the sender, in which each key only lives for a certain time interval and will be used in reverse order. While sending broadcast packets, the sender uses the currently active key to generate the

message authentication code (MAC). Later, the sender will disclose the key used before so that the receiver can verify the previously received packets, which is the so-called time-delayed key disclosure. In this case, receivers can ensure that the received data really originates from the correct source.

2.3 Trusted Execution Environment

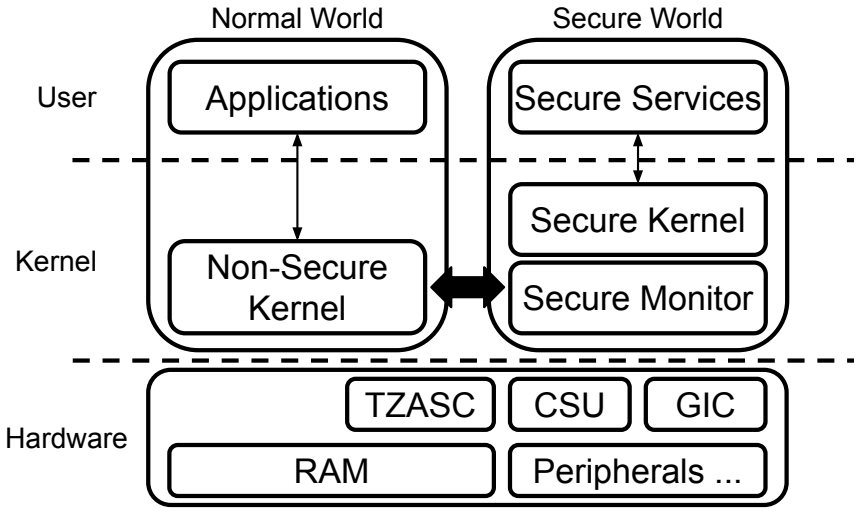


Figure 2.3: ARM Trustzone's secure monitor is responsible for directing call's to the secure world to the corresponding secure services and resuming execution in the normal world once it has finished.

ARM TrustZone is a hardware-enforced isolation technique enabled on ARM Cortex processors [14]. As shown in Figure 2.3, It creates two isolated execution environments: a trusted execution environment (TEE) and a rich execution environment (REE). TEE contains privileged permissions and can access reserved memory regions. The TrustZone Address Space Controller (TZASC) [12] provided by ARM TrustZone can partition portions of memory such that they are available only to the secure world. Besides, i.MX provides a TrustZone compatible component, the Central Security Unit (CSU), that extends the secure/non-secure access permission to peripherals. The CSU can be used to enable secure-only access for different peripherals. Any invalid accesses will result in an asynchronous external abort exception, similar to a device interrupt. With the pres-

ence of TEE, asynchronous hardware interrupts can also be routed directly into the secure world, which allows sensors and CV network interface to map all their interrupts to the secure world.

Regardless of privilege, the normal world processes cannot access the instructions and memory in the secure world. In order to execute secure instructions, the normal world must trigger a context switch to the secure world using a special `smc` instruction, which generates a synchronous exception and suspends execution in the normal world [13]. The secure monitor handles the exception; then, the secure world is activated for execution.

OP-TEE. OP-TEE [126] is an open-source implementation of TEE, which usually works with a non-secure Linux kernel running on ARM. OP-TEE implements TEE Internal Core API v1.1.x, which is the API exposed to Trusted Applications, and the TEE Client API v1.0, which is the API describing how to communicate with a TEE. Those APIs are defined in the GlobalPlatform API specifications [60].

2.4 CAV System

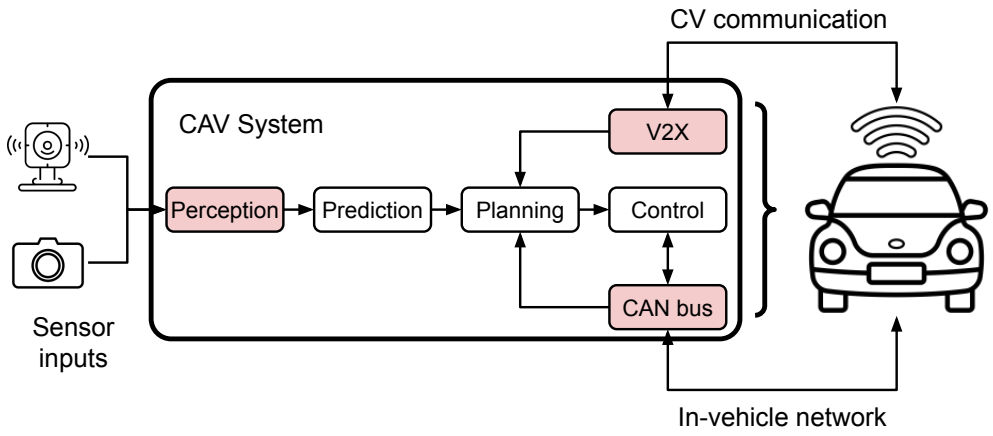


Figure 2.4: Overview of the CAV system.

The CAV consists of multiple functional modules, as depicted in Figure 2.4. The perception module is usually the first component in the pipeline, which processes raw sensor data from various sensors (e.g., Camera, LiDAR). Then, ML-based object detection approaches are adopted to

precept the surrounding obstacles. Afterward, the prediction module predicts future trajectories for observed obstacles, based on the historical location information. With the predicted trajectory information and the current vehicle status, the planning module calculates feasible trajectories for the CAV. At last, control commands are transmitted to the CAN bus to control the vehicle's behavior. Additionally, the V2X module is an emerging module that incorporates the CV communication capability. Therefore, the V2X module can directly provide the location and speed information of the surrounding vehicles.

Inevitably, the CAV system introduces new attack surfaces (i.e., red boxes in Figure 2.4). Among these functional modules, the perception module, the V2X module, and the CAN bus are the most vulnerable ones, as they directly accept critical data from the external environment. For example, the attacker can intentionally transmit falsified data through the CV network.

2.5 Related Works

We summarize the related works in several categories below.

CV security analysis. Since the idea of VANET (i.e., the original idea of CV) has been around for more than ten years, many researches have already studied general threats to CV network [226, 5, 117, 173, 173, 75, 78]. However, existing works generally suffer from three limitations. First, they rely on manual inspection to identify potential threats [117, 173, 226], as opposed to automatic discovery in our work. It is also hard to automate the risk assessment of identified threats in these works. For example, Laurendeau et al. [117] use ETSI's threat analysis methodology [55], which relies on human to qualitatively rank the risks of the threats. Similarly, Petit et al. [173] manually characterized threats in the automated vehicle (e.g., the cooperative automated vehicle with V2X communication), only annotated the qualitative risk.

Second, they focus on security properties such as integrity, confidentiality, and privacy [172, 29, 212, 5, 75], as opposed to availability in our work. Although USDOT and the protocol designers have already employed security mechanisms to protect the integrity and confidentiality of CV

network communication [212, 5], the protocol stack may still suffer from availability issues. For instance, if an incoming packet that may result in a safety threat cannot pass the verification, it will be discarded without triggering any warnings, and the application will not be able to process any incoming packets. To the best of our knowledge, only one prior work inspected the threats to availability [226], but it suffers from the third limitation below.

Third, they focus on prior generations of protocols or are conducted before the standardization of IEEE 1609 [78, 226, 5, 117, 173, 22], as opposed to the latest version studied in our work. For instance, the latest version of IEEE 1609.3 has integrated WAVE Service Advertisement (WSA) security considerations [226], in which Whyte et al. identify threats to availability of WSA in IEEE 1609.3-2010 [90] due to misconfigurations or malicious WSA access parameters.

Model checking security protocols. Model checking is a mature formal verification technique for finite state concurrent systems, and has been applied to several complex network protocols [146, 70, 150, 53, 82]. These works aim at exposing vulnerabilities in network protocols but does not consider quantitative assessments. CVANALYZER can finish the attack discovery and the quantitative threat assessment without touching implementation details. Therefore, CVANALYZER can be used by the protocol designer to evaluate the correctness of the protocol and also understand the severity of identified attacks, which can further guide the design of mitigation solutions. Also, this can largely minimize the cost to fix vulnerabilities, as all problems can be solved at the early stage.

Secure membership management. For a wireless ad-hoc network, a secure membership management system is necessary. Usually, network nodes form a peer group to share data with each other [149, 239, 181], and a group leader or other trusted entity is responsible for membership management. Wagner et al. [218] designed a decentralized blockchain-based system membership management for the platoon, in which each platoon member maintains a local copy of the blockchain, storing platoon information; however, it has scalability issues.

Due to the high mobility, the latest CV network does not form different communication groups, but adopts the digital signature (ECDSA), with the support of a PKI system, SCMS [29], to secure

the communication. Any CV devices with valid certificates can broadcast data to others. To manage membership, the recently deployed SCMS [29] introduces misbehavior detection to identify malicious or malfunctioning members and then revoke their certificates. For our attacks, the certificate revocation in existing SCMS cannot prevent P2PCD attacks but can mitigate PMP attacks. The learning response in *N1* and *N2* does not require any signing certificates, so the certificate revocation cannot prevent the attacker from launching these two attacks. In *N3* and *N4*, the attacker can always generate new syntax-valid certificates for the learning request (i.e., an SPDU). Since the vehicle cannot distinguish the self-generated certificates with unknown certificates, the learning request field will still be processed. Unless the vehicle can always connect to the PKI (through RSU) to check the validity of unknown certificates, it is impossible to prevent the attacker from using self-generated certificates in the current CV network stack. Unfortunately, communication with the infrastructure may not always be present due to the deployment difficulties. We admit that if the PKI supports the online certificate status check, with the infrastructure coverage increase, the impact of P2PCD attacks will be diminished.

Security analysis for in-vehicle protocols. Zelle et al. [232] have recently studied the applicability of TLS in the resource-constrained in-vehicle network. Lastinec et al. [116, 115] have also investigated the possibility of applying IPsec for securing in-vehicle communications. However, they mainly focus on performance evaluation and do not develop a solution to ensure source authentication. Moreover, Lauser et al. [118] utilize Tamarin prover [142] to analyze the authenticity of AUTOSAR's Secure Onboard Communication (SecOC) protocol. There are also a list of works that uses formal analysis to analyze security protocols, such as TLS 1.3 [49, 48], 4G LTE/5G [82, 21, 47, 83], WPA2 [50], IoT protocols [106], and V2X protocols [77].

Source authentication. Source authentication [19] is not a new topic. To ensure source authentication in an efficient way, TESLA [169, 168, 170] uses symmetric cryptography with time-delayed key disclosure, which introduces extra delay for the broadcast communication. In addition, they do not consider the real-time requirements of in-vehicle communication. To accommodate the performance requirements of the in-vehicle communication, GATEKEEPER does not follow TESLA,

but introduces an on-path authenticator to ensure the source authentication with low overhead.

Besides, Origin and Path Trace (OPT) [107] and Passport [133] are two protocols that provide source authentication and path authentication properties. Both of them are designed for Autonomous Systems (ASes) at the Internet scale. However, we cannot directly adopt them for the in-vehicle network. First, they only work for two-parties communication. Second, their design does not consider the resource constraints and thus result in high overhead at the sender side, violating our design goals. Groza et al. [64] propose LiBrA-CAN for in-vehicle broadcast authentication, while a LiBrA-CAN sender needs to more number of MAC operations than GATEKEEPER.

Protocol	Time overhead (# MAC operations)			Bandwidth overhead (# MACs)
	Sender	Receiver	Intermediate devices	
OPT [107]	$(M+1)R$	2	$2R$	$(M+1)R$
Passport [133]	$(M+1)R$	1	R	$(M+1)R$
LiBrA-CAN [64]	$\binom{N-1}{G-1}$	$\binom{N-2}{G-2}$	$\binom{N}{G} - G$	$\binom{N-1}{G-1}$
GATEKEEPER	2	2	R	2

Table 2.1: Comparison with OPT [107], Passport [133], and LiBrA-CAN [64]. R: the number of receivers. N: the total number of participants, including both senders and receivers (i.e., $N = R + 1$). M: the number of intermediate devices (e.g., routers for OPT and Passport). G: the group size.

In Table 2.1, we compare GATEKEEPER with OPT [107], Passport [133], and LiBrA-CAN [64]. The intermediate devices represent routers for OPT/Passport, the master node for LiBrA-CAN, and the authenticator for GATEKEEPER. OPT and Passport are not designed for broadcast communication, so we adapt them to the broadcast scenario and multiply the corresponding time and bandwidth overhead with the number of receivers M. Obviously, in this case, the time and bandwidth overhead of OPT and Passport are higher than GATEKEEPER. Notably, OPT and Passport can also ensure path authentication, which is one source for the relatively high overhead.

LiBrA-CAN divides all communication participants into different groups with a group size of G. The sender needs to generate $\binom{N-1}{G-1}$ MACs for all groups that he or she is part of. Therefore, the time overhead of the sender is always greater than or equal to $N-1$ (i.e., $\binom{N-1}{G-1} \geq N-1$).

Similiar bounds can be derived for the time overhead of receivers and intermediate devices as well as bandwidth overhead, and all of them are higher than GATEKEEPER.

TEE-based system. As mentioned in § 2.3, TEE provides isolated execution environments; thereby, many researchers choose to protect security-sensitive programs inside the TEE. Such TEE-based systems are widely appeared in different domains, such as mobile applications [187, 128, 204, 120], edge computing [165, 163], cyber-physical systems (CPSs) [145, 221].

Trusted Language Runtime (TLR) [187] provides a small runtime engine for .NET mobile applications inside the TruztZone TEE. The mobile applications must be partitioned into security-sensitive and insensitive parts. However, the TLR does not offer access to peripherals in the secure world and relies on the normal world for peripheral interactions. VeriUI [128] uses TrustZone to construct a tamper-proof environment for entering and validating users' passwords. TrustOTP [204] utilizes TrustZone to isolate the One-Time-Password (OTP) generation. Both VeriUI and TrustOTP include a secure touchscreen driver and a secure display controller in the secure kernel for user interactions. Lentz et al. [120] propose SeCloak, which focuses on peripheral management. SeCloak allows users to enforce secure and verifiable control over peripheral availability (on/off).

For secure edge computing platforms, Part et al. [165] propose StreamBox-TZ, a secure stream analytics engine for the TrustZone TEE on an edge platform. StreamBox-TZ only wraps a trusted data plane in the TEE to ensure high throughput and low delay properties.

For CPSs, the performance and availability of the system must be considered, in addition to the security requirements. Mishra et al. [145] present TEECheck, an on-device message checking system, which protects low-end embedded devices, like ECUs, in the automotive environment. TEECheck incurs low and predictable overhead and does not consume extra bandwidth in the CAN bus. Wang et al. [221] design and implement RT-TEE to protect safety-critical CPSs. Except for the integrity and confidentiality properties, RT-TEE focuses on system availability, leveraging a two-layer policy-based event-driven hierarchical scheduler.

However, the research works discussed so far all require manual modifications to the original

programs so that they can benefit from the TEE. TrustShadow [65] only introduces a runtime system in the TEE, which intercepts low-level exceptions, forwards them to the Linux kernel in the normal world, and verifies the return values. By doing so, TrustShadow can protect unmodified security-sensitive applications from the untrusted normal world.

Automated partitioning of security-sensitive programs. On the other hand, to avoid the human efforts of re-engineering the security-sensitive programs, many developers propose automated partitioning approaches to split the original programs into security-sensitive and security-insensitive parts. Only the security-sensitive part will be relocated to the TEE, reducing the TCB size. It is worth noting that prior works have studied the program partitioning of other types of applications for security policy enforcement or privilege separation [6, 31, 40, 229].

Unlike TLR [187], Rubinov et al. [186] develop an automated partitioning framework for Android applications. The framework generates code fragments relevant to manipulating confidential data, with privileged instructions for the TEE. However, human efforts are required to translate the extracted Java code fragments into C code in the TrustZone TEE. A similar system named Glamdring [127] is also proposed for Intel's SGX. Glamdring utilizes static dataflow analysis and static backward slicing to find all functions accessing and modifying the sensitive data. Also, Civet [209] focuses on Intel's SGX and generates partitioned programs for Java applications. Differently, Civet further hardens the enclave boundary to protect against malicious inputs. Besides, the authors of Civet develop a lightweight JVM and optimize the garbage collection for enclaves.

Program analysis to assist the program partitioning. SVF [202] is a widely used inter-procedural static value-flow analysis tool for C and C++ programs. However, SVF is a whole program analysis, rendering the analysis unscalable to large-size programs (e.g., Linux kernel) [235]. Instead, PtrSplit [132] presents a modular way of performing the whole program analysis to construct the program dependence graph (PDG). It only requires an intra-procedural pointer analysis, instead of a global pointer analysis like SVF. The core technique of PtrSplit is called *parameter trees*, which can propagate the intra-procedural pointer analysis results inter-procedurally. Therefore, PtrSplit can be used to partition user-space programs efficiently. KSplit [81] further extends

PtrSplit to the kernel-space programs to isolate device drivers in the kernel.

Device driver record and replay. Since many security-sensitive programs have to access the secure peripherals in TEE-based systems, the corresponding device drivers are needed in the TEE. Instead of including the whole device drivers, RT-TEE [221] debloats the original drivers by recording and replaying device/driver interactions to minimize the TCB size. Notably, RT-TEE is built upon the predictability of CPSs to convert complex device/driver interactions to the simple replay of a fixed set of I/O interactions. Similarly, GPUReplay [164] also records and replays the device/driver interactions, but only for GPU devices. CODY [163] further extends the recorder to a distributed fashion. To generalize the record and replay approach, Guo et al. [66] propose a holistic approach to generalize and parameterize recordings for a wide set of peripherals, deriving minimum viable device drivers for the TruztZone TEE.

Related Works	Fuzzing Inputs (controllable?)	Fuzzing Targets (whole system?)	Methods			Vulnerability Types
			Gray-box fuzzing	No simulator	Others	
AV-FUZZER [122]	Driving scenarios (✗)	Apollo (✓)	✗	✗	-	Traffic violations
ASF [79]	Driving scenarios (✗)	Apollo (✓)	✗	✗	Trajectory coverage metric	Traffic violations
AutoFuzz [238]	Driving scenarios (✗)	Apollo (✓)	✗	✗	NN-guided seed selection	Traffic violations
FusED [237]	Driving scenarios (✗)	OpenPilot (✓)	✗	✗	Fusion error definition	Fusion errors
VulFuzz [148]	OpenPilot messages (✗)	OpenPilot modules (✗)	✓	✓	Grammar mutation + directed fuzzing	Low-level system crashes
PlanFuzz [220]	Road objects (✓)	Planning module of Apollo/Autoware (✗)	✓	✓	Object location mutation + directed fuzzing	Semantic DoS of planning

Table 2.2: Comparison with existing CAV fuzzing works.

CAV fuzzing. Table 2.2 presents the comparison between existing CAV fuzzing works. CAVFUZZER in Chapter 6 is complementary to existing works because of different analysis goals. Existing CAV fuzzing works rarely consider the exploitability of identified vulnerabilities. The first five research works [122, 79, 238, 237, 148] take driving scenarios or internal messages of the CAV system as fuzzing inputs, which cannot be directly controlled by the attacker. Thus, the identified vulnerabilities are not exploitable for the attacker. We argue that it is crucial to consider exploitability. Such considerations can further help developers prioritize the high-risk vulnerabilities (i.e., exploitable vulnerabilities) and avoid being overwhelmed by the analysis results. To the best of our knowledge, PlanFuzz [220] is the only existing work that considers exploitability. In contrast, our work, CAVFUZZER, has a more comprehensive threat model than PlanFuzz and

employs a novel object-level mutator.

CHAPTER 3

Systematic Detection of Design-Level Flaws in CV Communication Protocols

3.1 Introduction

With the emerging Connected Vehicle (CV) technology [211], Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) wireless communication enables vehicles to exchange important safety and mobility information with other entities in real time. In September 2016, the U.S. Department of Transportation (USDOT) launched the CV Pilot Program in three sites, New York City, Wyoming, and Tampa, to spur early CV technology deployment and test CV safety applications in the real world. As of Fall 2018, the program has entered the third phase, which requires at least 18-month period for long-term operation and key performance measurements [213].

While CV technology can greatly benefit transportation mobility and safety, such dramatically increased connectivity inevitably increases the attack surface of both vehicles and the transportation infrastructure. For example, if the CV communication protocol stack is not sufficiently secure, attackers can directly cause safety hazard to human drivers on the road [1, 131, 34]. Thus, it is imperative to understand the potential security vulnerabilities in the CV network stack as early as possible so that they can be proactively addressed before large-scale deployment. To achieve this, it is necessary to start with a systematic study of potential design-level security flaws in the CV network stack, since both the discovery and defense solutions of such flaws can most generally affect the security of their corresponding implementation instances.

Existing work on the analysis of Vehicular Ad-Hoc Network (VANET) or CV security [1, 226, 5, 117, 173, 173, 75, 78, 22] generally suffer from three limitations:

(L1): they lack systematic approaches and rely on manual inspection to identify potential threats [117, 173, 226, 1], which is both insufficient and inefficient. It is also hard to automate the risk assessment of identified threats in these works. For example, Laurendeau et al. [117] use ETSI's threat analysis methodology [55], which relies on human to qualitatively rank the risks of the threats. Similarly, Petit et al. [173] manually characterized threats in the automated vehicle (e.g., the cooperative automated vehicle with V2X communication), only annotated the qualitative risk.

(L2): The threats to the availability of the higher-layer protocols (i.e., IEEE 1609 protocols [91, 93] and CV applications), which can prevent legitimate protocol participants from accessing critical services in the network, are largely under explored [172, 29, 212, 5, 75]. Although USDOT and the protocol designers have already employed security mechanisms to protect the integrity and confidentiality of CV network communication [212, 5], the protocol stack may still suffer from availability issues. For instance, as shown in Figure 3.1, if an incoming packet that may result in a safety threat cannot pass the verification, it will be discarded without triggering any warnings, and the application will not be able to process any incoming packets. To the best of our knowledge, only one prior work inspected the threats to availability [226], but it suffers from the third limitation below.

(L3): Previous works mostly target prior generations of the protocols, ignoring the analysis of CV applications, or are conducted before the standardization of IEEE 1609 protocol family, and hence some discovered vulnerabilities do not exist in the latest CV network stack design [78, 226, 5, 117, 173, 22]. For instance, the latest version of IEEE 1609.3[1] has integrated WAVE Service Advertisement (WSA) security considerations [226], in which Whyte et al. identify threats to availability of WSA in IEEE 1609.3-2010 [90] due to misconfigurations or malicious WSA access parameters.

In this chapter, we perform the first rigorous security analysis to automate the discovery of

[1]In the following text, without specific notations, "IEEE 1609.*" represents the latest version (e.g., "IEEE 1609.2" and "IEEE 1609.3" stands for "IEEE 1609.2-2016" and "IEEE 1609.3-2016" respectively).

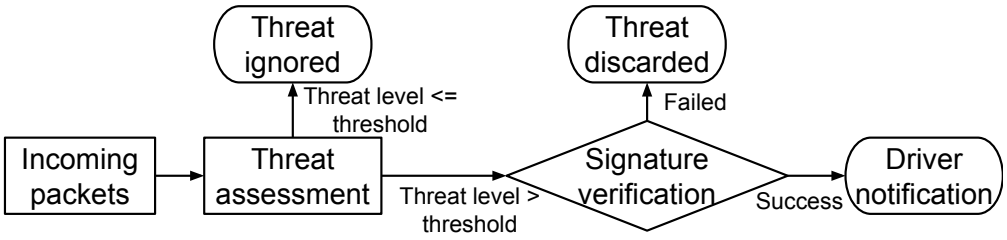


Figure 3.1: Verify-on-Demand [5, 110]: a connected vehicle will only verify the signatures of incoming packets, if packets result in a safety threat level above the threshold.

availability or DoS (Denial of Service) vulnerabilities, in (1) the latest version of the IEEE 1609 protocol family and (2) Cooperative Adaptive Cruise Control (CACC) applications. To address L1 (i.e., manual analysis), we formulate the analysis as a model-checking problem and design a novel system, CVANALYZER, that leverages (1) a general model checker (MC) [231] and (2) a probabilistic model checker (PMC) [111] to automate *both* the attack discovery and the attack assessment. Either model checker alone cannot achieve our analysis goal [20]. MC [42, 231, 74, 52] is useful for the attack discovery [146, 70, 150, 53, 82, 21]; while, for tractability reasons, PMC (e.g., PRISM [111]) has limited support in finding vulnerabilities and mainly focuses on quantitative property verification. Therefore, we utilize MC and PMC to verify availability-related properties and quantitative properties respectively.

To address L2 (i.e., no availability threat analysis), we define security properties to cover both availability-related properties (e.g., "all CV devices should eventually learn unknown certificates") and quantitative properties (e.g., "what is the expected time delay of processing next packet?"). By verifying these properties, we not only identify potential vulnerabilities but also understand the corresponding security consequences.

To address L3, we inspect the *latest* specifications [214] of the CV network protocols and one complicated CV application (i.e., CACC). For the former, we focus on *newly* added *CV-specific* features (e.g., P2PCD); for the latter, we pick two platoon management protocols (PMPs) (VENTOS [217, 8] and PLEXE [174, 192]), which are widely used by researchers, practitioners, and developers. We choose to study PMP because (1) high importance, since it can directly control vehicles and thus impact safety [1, 54], and (2) high demand for systematic verification, since it

involves distributed collaboration among multiple vehicles and thus highly difficult to effectively analyze using only manual efforts. We abstract the CV protocols as multiple finite state machines (FSMs). In the abstract model, each FSM represents a protocol participant, and all participants communicate with each other through adversary-controlled public communication channels. Notably, such abstract model ignores the low-level implementation details, which is suitable for finding design flaws.

By design, CVANALYZER does not trigger any false positives, aiming to guarantee *soundness*. That is, if we report a property violation, it is indeed a violation; we cannot, however, detect all violations. Like existing works on model checking security protocols [146, 70, 82], our analysis is parameterized by the number of protocol participants. Given a specific number of protocol participants and a set of properties, model checking guarantees to exhaustively enumerate all reachable states. Therefore, a model checker should also have *completeness*, i.e., if the model checker does not report any property violations, then the model is proved to be correct. However, due to the undecidability of parameterized system verification problem [11], achieving both soundness and completeness is impossible, and we cannot enumerate all possible number of protocol participants. In this case, we follow the conventional method of aiming for soundness instead of completeness.

In model checking, the model size (i.e., the total number of reachable states) grows exponentially with the number of state variables and the number of protocol participants. To alleviate the state explosion [44] problem in applying model checking to complex network protocols, we propose an *abstraction* approach (§ 3.3), which reduces unnecessary state variables and merges a large data domain into a small equivalent data domain. We ensure that our state reduction approach does not introduce wrong property violations (i.e., false positives).

Overall, our contributions are summarized as follows:

- We perform the *first* rigorous security analysis to find DoS attacks in the *latest* version of IEEE 1609 protocol family and two PMPs via the model checking technique. To achieve this goal, our analysis methodology design aims at providing soundness without triggering any false positives. To alleviate the state explosion problem, we propose a novel *abstraction*

approach, which does not generate any false positives and can also achieve complete model coverage.

- Using CVANALYZER, we are able to discover 4 *new* DoS vulnerabilities in P2PCD, which can block the certificate learning process and can further prevent the application layer from processing incoming packets, and 15 vulnerabilities (14 of 15 are new) in PMPs, which can block the communication among platoon members. Our quantification results show that their exploits can have as high as 99% success rates, and can double the delay in packet processing, which violates the latency requirement of CV communication.

- For these newly-discovered vulnerabilities, we have constructed practical exploits and validated them in a real-world testbed. We have also reported to and received confirmations for P2PCD attacks from IEEE 1609 Working Group [94]. Besides, our case studies demonstrate that P2PCD attacks can lead to traffic accidents, and PMP attacks can affect the speed stability of the victim vehicle. These results thus concretely demonstrate the effectiveness of CVANALYZER.

- For the identified vulnerabilities, we discuss the fundamental reasons and propose effective mitigation solutions, including avoiding using truncated hash value (e.g., 3-byte hash value), mandating verification for P2PCD learning responses, and requiring P2PCD learning requests to be broadcast (§3.6). After our discussion with the IEEE 1609 Working Group [94], mitigation solutions against P2PCD attacks are planned to be integrated into the next version of IEEE 1609.2.

3.2 Threat Model

CV communication capability. In our work, we assume that the attacker can compromise OBUs on her own vehicles or others' vehicles, which follows recent works on CV security [37, 38, 227]. This assumption is reasonable, as previous works [34, 108] have already shown that in-vehicle

systems can be compromised physically or remotely. In this case, the attacker can send malicious packets to other vehicles through compromised CV devices. All malicious packets should comply with protocol specifications. Notably, the attacker is allowed to unicast malicious packets to a specific vehicle (cf. IEEE 1609.3, Subclause 5.5.1).

Passive monitoring. The attacker can passively eavesdrop and capture all network traffic in her wireless communication range under the promiscuous mode of the wireless adapters.

Cryptography operations. We assume that cryptography operations used in CV protocols (e.g., signing, verification, and hash) are secure. The attacker thus cannot forge digital signatures used for packet authentications but can use valid certificates installed in compromised vehicles to sign outgoing packets. However, the attacker can still (1) passively collect valid certificates by sniffing the CV network traffic, and (2) construct local certificates, which are not signed by trusted anchors.

3.3 Analysis Methodology

In this section, we first present our how we construct each component in the model, including the adversary model and each protocol state machine. We then describe how we reduce the state space and document how we implement CVANALYZER.

3.3.1 Model Construction

As shown in Figure 3.2, our model, consisting of the environment and protocol state machines ($\mathcal{P}$), is driven by network and timer events. In general, the environment manages packet/time events generated by protocol state machines. It delivers triggered events (e.g., packet reception, timeout) to protocol state machines.

Adversary-controlled communication environment. We follow the design in prior works [182] and define three sequential steps in a loop for the environment:

1. `Retrieve`: the environment picks one of many different packet/time events if such an event

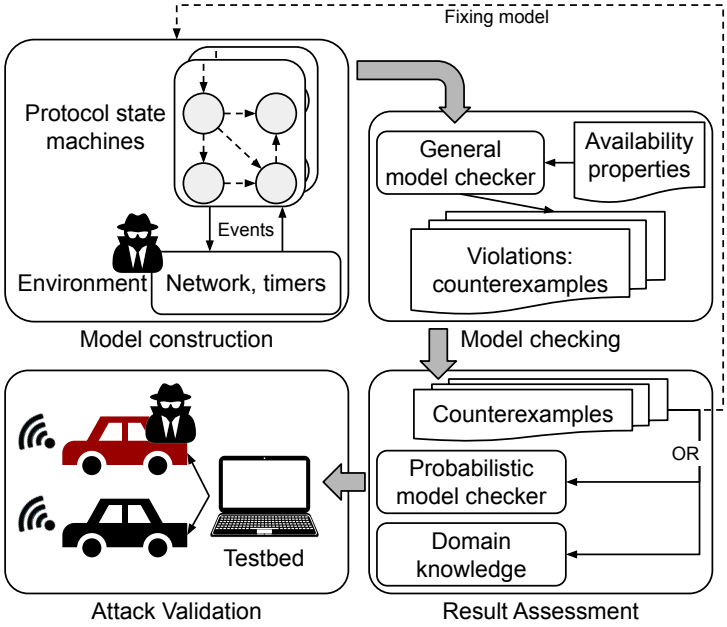


Figure 3.2: CVANALYZER overview. (Events: (1) incoming/outgoing packets, (2) added/deleted/-expired timers)

is available.

2. `Process`: the protocol state machine processes an event.

3. `PostProcess`: after processing a given event, the protocol state machine either sends a new packet, adds a new timer, cancels an existing timer, or does nothing. The environment needs to update its internal states and keeps track of newly added events.

Our threat model (§ 3.2) assumes that the attacker has communication and eavesdropping capabilities. Thus, we add one more step for the attacker to send and receive arbitrary packets:

4. `Attack`: the attacker is able to monitor all packets in the environment. If needed, she can inject arbitrary packets into the environment, which allows a protocol state machine to process all possible packet events.

To model the network, we construct the communication channel $\mathcal{C} = \{\mathbf{ch}_{i,j} | i, j \in [1, n], i \neq j\}$, where $\mathbf{ch}_{i,j}$ is a FIFO queue from $\mathcal{P}_i$ to $\mathcal{P}_j$. In this case, the packet sending and reception are abstracted as enqueue and dequeue operations on $\mathbf{ch}_{i,j}$. Notably, we do not consider network

factors for vulnerability discovery, such as network latency and packet loss, because the lossy and erroneous network weakens the attack's capability and increases the complexity of the model. Placing the attacker in her best position can help us uncover all potential attacks. On the other hand, to model timers, we do not keep track of the absolute time but only care about the temporal ordering of events, which is a common practice in model checking distributed system [112]. For progress advancing, all timers will count down simultaneously if there are no active events that should be delivered to protocol state machines.

Protocol state machine. All protocol participants ($\mathcal{P}_i$, $i \in [1, n]$) are identical; therefore, each of them can be represented as the same finite-state machine (FSM). Then, our model $\mathcal{M}$ can be defined as a concurrent system $\mathcal{M} = \mathcal{C}||_{i\in[1,n]}\mathcal{P}_i$, including an adversary-controlled environment $\mathcal{C}$ and n isomorphic processes $\mathcal{P}_i$, where $||$ is commutative and associative.

In our analysis, we abstract the higher-layer protocols in the CV network stack: (1) the communication model defined in networking services and message sublayer, (2) security services, and (3) PMP described in [8, 192, 217, 174]. We follow their specifications or codebases to define packet and timer handlers, which update the internal states of $\mathcal{P}_i$ while processing packets and timeouts delivered by the environment. Our model excludes the handler of certificate revocation in security services, because it relies on an external public key infrastructure (PKI) like SCMS [29] to revoke certificates, which is out of the scope of the network stack itself. How SCMS affects identified vulnerabilities is discussed in § 2.5.

For the security services, we first abstract away cryptographic constructs because we assume that the cryptography operations in CV protocols are secure. Then, we model both packet type and packet header data, as they are required by the internal security mechanisms. In CV network, each protocol participant will have a batch of unique end-entity certificates (a.k.a., signing certificates). To trigger all internal security mechanisms, for the certificate configuration, we assume that the issuer of each batch of signing certificates is different from each other and is attached with packets in transmission.

Probabilities. Network protocol involves many concurrent events (e.g., packet transmission),

leading to concurrent transitions in state machines. While building probabilistic models, we develop a discrete-time Markov chain (DTMC) model that assigns uniform probabilities to concurrent state transitions, originating from the same state (§ 3.3.2).

State reduction. We now show how we abstract the model to reduce states through a concrete example. For ease of exposition, we rely on a simplified example (Figure 3.3) derived from `N4` (§ 3.4.1.3). Our goal is to reduce unnecessary states to get an abstracted model. Also, we want to ensure that *the counterexample found in the abstracted model is a valid counterexample in the original model.*

```
1  h(x) ≜ x%M  h(x) = x mod M
2  EventRange ≜ (0 .. (N − 1))
3  TimerIndexRange ≜ (0 .. (M − 1))
4  Init ≜  Initial state
5      ∧ event ∈ EventRange
6      ∧ timer = [i ∈ TimerIndexRange ↦ None]
7  Next ≜  Specify how to update states
8      ∧ event′ ∈ EventRange
9      ∧ timer′ = [i ∈ TimerIndexRange ↦
10         IF h(event) = i THEN TIMEOUT  initialize the timer
11          ELSE IF timer[i] = None THEN timer[i]  not initialized
12                ELSE IF timer[i] > 0 THEN timer[i] − 1  count down
13                     ELSE None]  expire
14 Property ≜
15     ∀ i ∈ TimerIndexRange :
16        (timer[i] = TIMEOUT) ⇝ (timer[i] = 0)
```

Figure 3.3: A simplified example derived from `N4` (`N`: the total number of events; `M`: the total number of timers; `TIMEOUT`: the maximum value of the timeout).

In the example, we develop a simplified protocol, in which the model updates the `timer` according to the `event` (Line 9-13), in which the function $h(x)$ abstracts the hash truncation operation in P2PCD. Assuming that, without the attacker, the range of $event$ is $[0, X-1]$, where $X < M \leq N$. The attacker in the environment can trigger all possible events $[0, N-1]$.

For a given $event$, if $h(event)$ equals to i, then $timer[i]$ will be initialized (Line 10). For

other unmatched timers, a timer will (1) remain unchanged if $timer[i]$ is not initialized ($None$), (2) count down if it has been initialized, or (3) set as uninitialized if it expires. To capture the attacker's behavior, for each $Next$ step in Figure 3.3, we randomly select a value in $EventRange$ as the next $event$ (Line 8). Notably, events within $[X, N-1]$ are triggered by the attacker and can lead to the initialization of all timers. Last, to find counterexamples, we specify a liveness property (Line 14-16) that all timers should eventually expire if it has been initialized.

Obviously, the state space of the model depends on N and M, which can be arbitrarily large. For example, the timer index range in P2PCD would be $[0, 2^{24}-1]$ (i.e., $M = 2^{24}$). The number of events N can be 2^{256}. Unfortunately, the model checker cannot handle such large state space.

By analyzing the model, we observe that we do not need to track all $timer[i]$, as the protocol only updates a small set of timers when no attacker is presented. As stated before, without the attacker, the range of $event$ is $[0, X-1]$; the model thus only updates $timer[i]$, where $i \in [0, X-1]$. Usually, the protocol instance does not care whether other timers can eventually expire. Therefore, apart from reducing $TimerIndexRange$ to $[0, X-1]$, we also derive a weakened property, $\widehat{Prop}$:

$$\forall i \in [0, X-1] : (timer[i] = TIMEOUT) \rightsquigarrow (timer[i] = 0)$$

Since $\neg\widehat{Prop} \Rightarrow \neg Prop$, our decision ensures that if the identified counterexample violates $\widehat{Prop}$, it also violates $Prop$ and is a valid counterexample in the original model.

On the other hand, we observe that many events triggers the same update on $timer$. For example, both $event = 0$ and $event = M$ leads to the initialization of $timer[0]$. Thus, we decide to keep a small set of $EventRange$. We first partition $EventRange$ into several equivalence classes:

$$EventRange_i = \{j \in EventRange | h(j) = i\}, i \in [0, M-1]$$

where every event in $EventRange_i$ triggers the same update on $timer[i]$. For each equivalence class $EventRange_i$, we then pick one value, $EventRange_i = \{i\}$, so that we can trigger all updates on $timer$. In this case, we reduce $EventRange$ to $[0, M-1]$. However, among this range, only events in $[X, M-1]$ is triggered by the attacker, meaning that the attacker itself cannot trigger all updates on $timer$. We thus enlarge $[X, M-1]$ to $[X, 2M-1]$ so that the attacker itself can

trigger the initialization of all timers. Finally, we derive a small $\widehat{EventRange} = [0, 2M-1]$ and a mapping function:

$$f(x) = \begin{cases} x, & x \in [0, M-1] \\ M+i, & x \in \{j \cdot M + i | j \in [1, \lceil \frac{N}{M} \rceil - 1]\} \end{cases} \quad (i \in [0, M-1])$$

Moreover, f is a surjective function; thereby, for every $\hat{x}$ in $[0, 2M-1]$, we can always find at least one x in $[0, N-1]$ such that $\hat{x} = f(x)$. In another word, for every identified counterexample in the abstracted model, we can always find at least one corresponding counterexample in the original model by applying the inverse function f^{-1} on $event$.

By combining the aforementioned two strategies together, we can successfully reduce the state space of the example and ensure no wrong property violations. In particular, we reduce $TimerIndexRange$ and $EventRange$ to $[0, X-1]$ and $[0, 2X-1]$ respectively.

3.3.2 Model Checking

The goal of using the general model checker is for vulnerability discovery. Given a model $\mathcal{M}$ and security properties, once the model violates a property, the general MC will generate a counterexample, an execution trace leading to the violation. Formally, a model can be defined as consisting in a finite set of states S, initial states $I \subseteq S$, the transition relation $T \subseteq S \times S$, and a labeling function from states to a finite set of atomic propositions $L : S \to 2^{AP}$ [43]. Table 3.1 summarizes the high-level properties to analyze P2PCD and PMPs. For each property, we first refine φ_i to get a new property $\varphi_i\prime$ such that $\varphi_i \Rightarrow \varphi_i\prime$ and $\neg\varphi_i\prime \Rightarrow \neg\varphi_i$. For example, a refinement over φ_1 would be *at least one CV device should eventually broadcast a learning request after observing an unknown certificate*. Then, MC is used to find property violations. By analyzing the counterexample, we can formulate the attack procedure (§ 3.4) and analyze the fundamental reasons for identified attacks, which is helpful for the mitigation design (§ 3.6). Last, we patch the model to ensure that the general MC will not generate the same type of violations later.

PMC aims at avoiding manual risk assessment and does not discard identified vulnerabilities from the general MC. It helps assess the severity of the exposed vulnerabilities and thus allows

ID	Availability properties
φ_1	The application layer should be always able to consume valid incoming packets.
φ_2	Refinement over φ_1: All CV devices should eventually learn unknown certificates.
φ_3	Refinement over φ_1: All platoon members should eventually switch to idle state.

Table 3.1: Availability properties used by CVANALYZER

ID	Quantitative properties
ψ_1	What is the success rate of the attack?
ψ_2	What is the expected time delay of processing next SPDU?

Table 3.2: Quantitative properties used by CVANALYZER to quantify the security consequences of *N1-4*

the protocol designers to prioritize the solution design. Unlike the general MC, PMC assigns probabilities for each state transition $T : S \times S \rightarrow [0, 1]$ such that $\forall s \in S : \Sigma_{s\prime \in S} T(s, s\prime) = 1$. Since we assign uniform probabilities to concurrent state transitions, for all reachable successor states of s in $Succ_s = \{s\prime \in S | T(s, s\prime) > 0\}$, the transition probability between s and any $s\prime$ is $\frac{1}{|Succ_s|}$. A transition matrix can be derived from the transition probabilities. Thus, PMC can calculate the likelihood of transitioning from initial states to any target states. If we can formalize the states of the attack success, PMC can help us generate the attack success rate. Apart from the probability, PMC can also assign "time" costs for state transitions, which can be used to quantify time-related properties. In § 3.4, we leverage PMC to quantify the severity of non-deterministic attacks *N1-4*, which are defined as attacks that may not always succeed per attempt. We observe that, P2PCD attacks can succeed, *only if* malicious packets are delivered to the victim vehicle exactly within the attack time window. However, the attacker cannot precisely infer the start and end of the time window, but only roughly predict the start time. Thus, we use PMC to quantify their severity based on the success rate and the time delay.

3.3.3 Implementation

Following the proposed approach, to instantiate CVANALYZER, we use TLC [231] as the general model checker due to its expressiveness of constructing the model, and pick PRISM [111] as our probabilistic model checker. As the prior step of model checking, we manually extract the abstract model of the IEEE 1609 protocol family [214] and PMPs [217, 174]. The abstract model includes two (i.e., $n = 2$) legitimate vehicles and one malicious vehicle (i.e., the attacker). Then, we need to implement concrete models in the modeling languages used by TLC and PRISM. As the supported maneuvers of PLEXE is a subset of VENTOS, we merge them together as one model. The properties that we want to verify are shown in Table 3.1 and Table 3.2, covering availability and quantitative properties respectively.

3.4 Analysis Results

In this section, we describe 4 DoS attacks in P2PCD and 15 attacks in VENTOS [217] and PLEXE [174] in detail (Table 3.3). Then, we analyze the security implications of identified attacks, and quantify the success rate and the average time delay in packet processing of those non-deterministic attacks.

ID	Name	Assumption	New?	Implications
N1	Response Mute	Known response threshold, optional response verification, enough computing power	Yes	Stop the CV device from sending learning responses; result in traffic accidents (§ 3.5.2.1)
N2	Request Mute	Optional response verification, enough computing power	Yes	Stop the CV device from sending learning requests; result in traffic accidents (§ 3.5.2.1)
N3		Known MAC address		
N4	Numb	Known MAC address	Yes	Stop the CV device from recording unknown certificates; result in traffic accidents (§ 3.5.2.1)
A1, *A2*	(Prerequisites)	Available platoon space	A1: No [1]. A2: Yes	Cause traffic collision [1], lead to *A3-15*
A3, *A4*	Split Trigger	Centralized platoon coordination	Yes	Interfere the traffic flow stability, decrease efficiency and safety (§ 3.5.2.2)
A5-14	PMP Block	-	Yes	Prevent platoon members from performing any maneuvers
A15	Inconsistency	Inappropriate validity check	Yes	Lead to failures of the split maneuver and the leader/follower leave maneuver

Table 3.3: Summary of attacks found in the CV protocols. (**N**: CV network protocol, P2PCD. **A**: CV application, PMP)

3.4.1 P2PCD Vulnerabilities

In summary, CVANALYZER finds 4 new DoS attacks that can compromise the availability of CV network. All 4 vulnerabilities come from P2PCD [91], which prevents victim vehicles from learning unknown certificates (see Figure 3.4). Without knowing necessary certificates, the victim vehicles cannot verify incoming packets; the CV network stack thus cannot deliver data to the application layer. Besides, we discuss the fundamental reasons for these vulnerabilities. Also, we assess their security consequences.

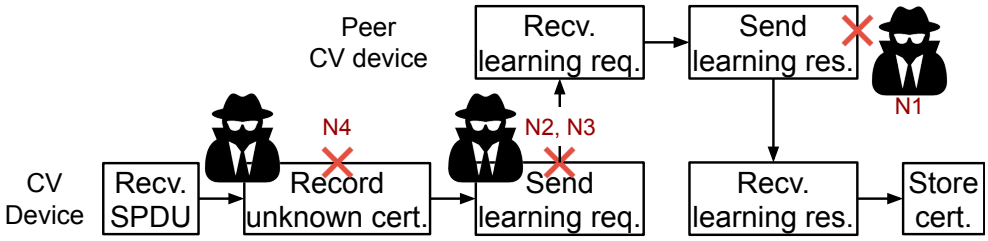


Figure 3.4: Four P2PCD attacks can break the whole pipeline of P2PCD learning process to prevent the CV device from learning/storing the unknown certificate.

In the following descriptions, two CV devices, `Vehicle 1` (`V1`) and `Vehicle 2` (`V2`), broadcast SPDUs every 100 *ms*. However, `V2` cannot verify packets sent by `V1` because `V2` does not know the issuer `ca1` of the signing certificate `ee1` used by `V1`. `V2` thus wants to learn the unknown certificate `ca1`. For each attack presented below, `V1` first sends a trigger SPDU to `V2`. In the normal case without the attacker, after receiving the trigger SPDU, `V2` initializes P2PCD learning process and attaches learning request information in the next outgoing SPDU. `V1` will construct and send the learning response after receiving the learning request.

3.4.1.1 Response Mute Attack

N1 can prevent a peer CV device from sending the learning response. This attack exploits the optional verification of learning responses and the throttling mechanism of P2PCD that limits the number of responses to a single request. The attacker intentionally interact with `V1` by sending multiple malicious learning responses to ensure that the response counter of `V1` exceeds the re-

sponse threshold. As a consequence, V1 choose not to send the learning response, and V2 fails in learning the unknown certificate ca1.

Assumptions. For successfully carrying out this attack, the attacker needs to know the exact value of the response threshold. For example, the response threshold of BSM is 3 [46]. We assume that V1 does not mandate the verification for incoming learning responses, which is consistent with the current protocol specification (§ 2.1.1). Also, we assume that the attacker has enough computing power to efficiently construct learning responses that can cause partial hash collision (e.g., low-order 3 bytes collision).

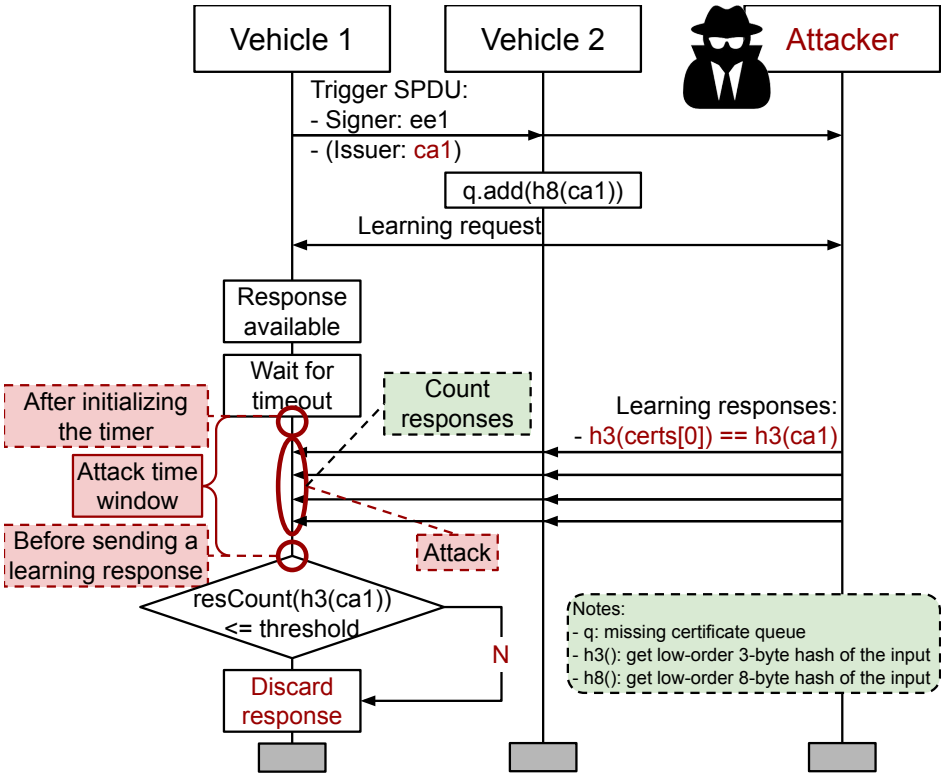


Figure 3.5: *N1*: the attacker can stop V1 from sending learning responses to V2 by sending multiple malicious learning responses.

Attack steps. Figure 3.5 illustrates the attack steps in detail. V1 first sends a trigger SPDU to V2. Instead of immediately sending the learning request, V2 stores the HashedId8 value of the unknown certificate ca1 in a queue (cf. IEEE 1609.2 [91], Subclause D.4.2.1.1). V2 attaches the HashedId3 value of ca1 in the learning request field of its next outgoing SPDU. In P2PCD,

`HashedId8` and `HashedId3` stands for the low-order 8-byte and 3-byte hash of a certificate respectively. After receiving the learning request, `V1` starts to prepare a learning response. Based on the throttling mechanism, `V1` initializes the response backoff timer and the response counter for the requested certificate.

However, the attacker can observe the trigger SPDU and the learning request, so she can determine that `V2` wants to learn an unknown certificate from `V1`. The attacker thus deliberately constructs multiple learning responses, in which the `HashedId3` value of the first certificate in the payload matches with the unknown certificate `ca1`. The attacker then sends out these malicious packets to saturate `V1`'s response counter (i.e., making it no less than the response threshold). On receiving malicious learning responses, `V1` wrongly updates its response counter (via `AddCertificate` primitive defined in IEEE 1609.2). When the response backoff timer expires, `V1` checks whether the response counter is less than or equal to the response threshold. Obviously, based on the current status of the response counter, `V1` decides to discard the response at this time.

Discussion. The reason for *N1* can be attributed to the use of truncated hash. By design, the hash function should be resistant to collision attacks. However, the use of truncated hash value compromises the security provided by the hash function. For example, for `HashedId3` used in CV network (i.e., three-byte hash), collision could be found in the brute-force number of 2^{24}. Most importantly, the response counter uses `HashedId3` as the identifier, which means that the attacker can manipulate the response counter if she constructs certificates leading to the partial hash collision. On the other hand, as introduced in § 2.1.1, IEEE 1609.2 does not mandate the verification for the learning response. Thus, it is still possible that some poorly implemented CV protocols may not verify the incoming learning response but just store certificates in the payload. Even if the CV device mandates the verification, the attacker can collect certificates with the attacker-desired hash values offline (§ 3.6). Note that, since P2PCD learning responses do not carry digital signatures, the attacker does not need to possess a legitimate certificate to launch *N1*, making the attack much more stealthy.

3.4.1.2 Request Mute Attack

Both *N2* and *N3* can stop CV device from sending learning requests. Similar to *N1*, *N2* exploits the hash collision issue. Readers can refer to Appendix A for more details.

N3 exploits the unicast capability and injects a malicious SPDU with the same learning request field (i.e., the `HashedId3` value of `ca1`) as what `V2` intends to send. As a result, `V2` can observe the malicious learning request and decides not to send its own learning request. `V2` hence fails in learning unknown certificate `ca1` because `V1` does not receive any learning requests.

Assumptions. To successfully launch this attack, the only requirement is that the attacker needs to know the MAC address of the victim vehicle `V2`. This is reasonable because the attacker can monitor all traffic in the network; it can thus observe `V2`'s MAC address from packets sent by `V2`.

Attack steps. As presented in Figure 3.6, `V2` initializes P2PCD after receiving a trigger SPDU from `V1`. `V2` stores the `HashedId8` value of the unknown certificate `ca1` in a queue. Meanwhile, since the attacker can observe the trigger SPDU, she constructs a malicious learning request, in which the learning request field `m.lr` equals to the `HashedId3` value of the unknown certificate `ca1`. In P2PCD, after receiving a learning request, `V2` removes any matching `HashedId8` entries in the queue. Therefore, `V2` removes the entry of the unknown certificate `h8(ca1)` in the queue, where `h8` is a function to get the low-order eight-byte hash of the input. As the queue becomes empty, `V2` decides not to attach the learning request information in the next outgoing SPDU. Consequently, `V2` is unable to learn the correct unknown certificate.

Discussion. The fundamental reason for *N3* is that once a vehicle observes an active P2PCD learning request, it will not send the learning request for the same unknown certificate. In the normal case, this mechanism is helpful to reduce the number of simultaneous learning requests in the fly. However, the attacker can unicast the learning request to the victim vehicle. Notably, the attacker should not send such learning request to the owner of the unknown certificate (i.e., `V1` in Figure 3.6). This attack misleads the victim vehicle to believe that some other legitimate vehicles are requesting the same unknown certificate. The protocol designers do not consider the use of unicast in P2PCD, which makes the victim vehicle vulnerable to *N3*. On the other hand,

N3 does not require the attacker to possess a legitimate certificate to sign the learning request but only uses self-generated certificates. As long as the digital signature of the learning request is valid, the vehicles will process the learning request field in the packet header. In this case, the signing certificate of the malicious learning request will be treated as an unknown certificate and will trigger another P2PCD learning process. Therefore, even if the certificates used by the attacker is revoked, the attacker can always generate new certificates for future use.

3.4.1.3 Numb Attack

First, like *N3*, this attack exploits the unicast capability and injects a malicious SPDU with the same learning request field (i.e., the `HashedId3` value of `ca1`) as what `V2` intends to send. This causes the same consequence as *N3*, in which `V2` chooses not to send the learning request and thus cannot learn the unknown certificate. Then, due to the request active timer (e.g., `reqActiveTimer`), `V2` still thinks that there should be an active request in the fly. Therefore, while receiving the next trigger SPDU, `V2` chooses not to add the `HashedId8` value of the unknown certificate `ca1` into the queue and keeps waiting for learning responses.

Attack steps. As described in Figure 3.6, this attack is similar to *N3*, but the attacker has different attack goal that it tries to prevent the victim vehicle `V2` from recording unknown certificates. Since `V1` broadcasts BSMs every 100 ms, `V2` will receive a trigger SPDU again in a few milliseconds. At this time, `V2` still cannot verify the incoming packet. However, because the request active timer has been initialized in the last communication round, and the timer is usually set to 250 ms [46], `V2` believes that there is still an active learning request in the fly. Thus, `V2` does not add anything into the queue, which means that it will not attach any learning request information in the next outgoing SPDU. `V2` cannot recover from this malicious state until the request active timer expires.

Discussion. *N4* has the same fundamental reasons as *N3*. The only difference is that the request active timer blocks the victim vehicle from recording unknown certificates in that the initial value (i.e., 250 ms) of the timer is around 3 times larger than the broadcast interval (i.e., 100 ms). Fortunately, P2PCD allows the user to configure the parameters for the initial value of timers.

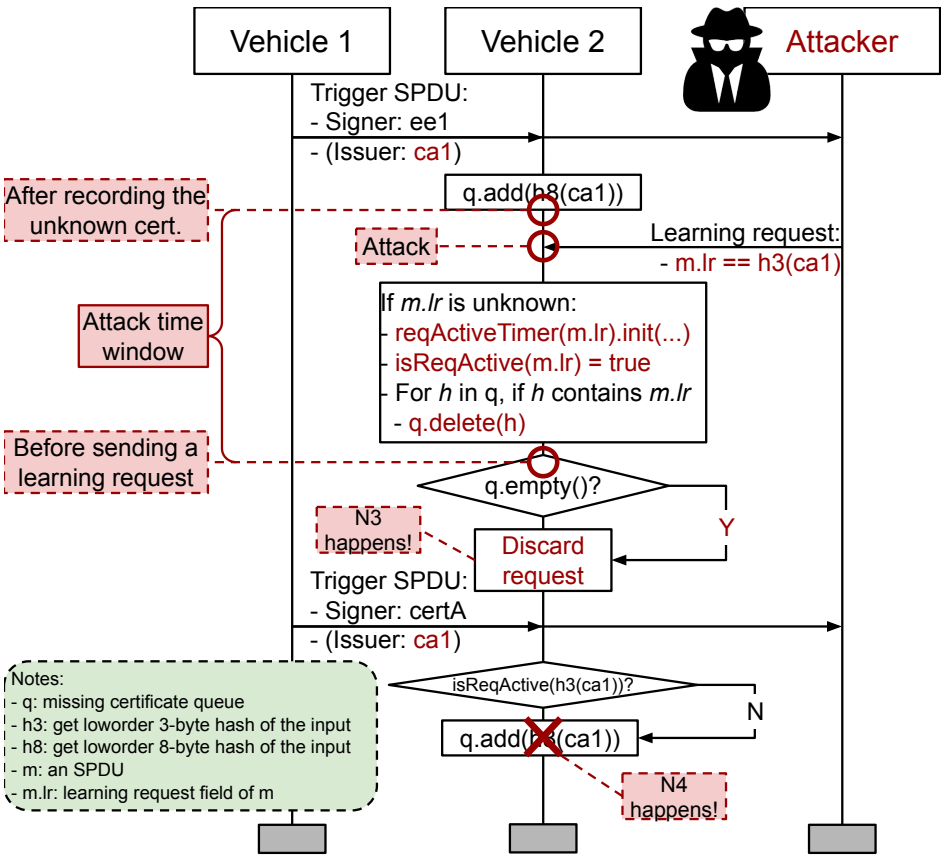


Figure 3.6: *N3* can stop V2 from sending learning requests to V1 by sending a malicious learning request.. *N4* can stop V2 from recording unknown certificates by sending one or more malicious learning requests.

3.4.1.4 Assessment

We observe that, *N1-4* can succeed, *only if* the attacker delivers the malicious packets to the victim vehicle exactly within the *attack time window*. However, one challenge for the attacker is that she cannot precisely determine the start or end of the *attack time window* but can only roughly estimate the time window. Thus, we are motivated to quantify the probability of successfully launching the attack by using the probabilistic model checker in CVANALYZER.

ID	Attack packet	Attack time window	Succ. Rate	Time delay (*ms*)
N1	RES-H3	0-250 ms	99.47%	580 (280 + 300)
N2	RES-H8	$\leq$ 100 ms	99.99%	370 (280 + 90)
N3&4	LR-H3	$\leq$ 100 ms	99.99%	570 (280 + 290)

Table 3.4: Attack assessment results of *N1-4*.

Table 3.4 summarizes the quantification results. Since *N3* and *N4* use the same type of packet to attack the victim vehicles, and the *attack time window* of them are the same, we merge these two attacks together and quantify the probability results based on the type of *attack packet*.

For *N1*, the success rate is 99.47%. We set the response threshold as 3 in our experiments. To successfully launch one attack, the attacker has at least send 4 malicious learning responses, while the rest attacks only need to send one malicious packet. This is why the success rate of *N1* is slightly lower than other three attacks. For *N2-4*, the success rates are 99.99%. If V2 is able to send the learning request before receiving the malicious packet, the attacker will fail. However, this is unlikely to happen based on our results.

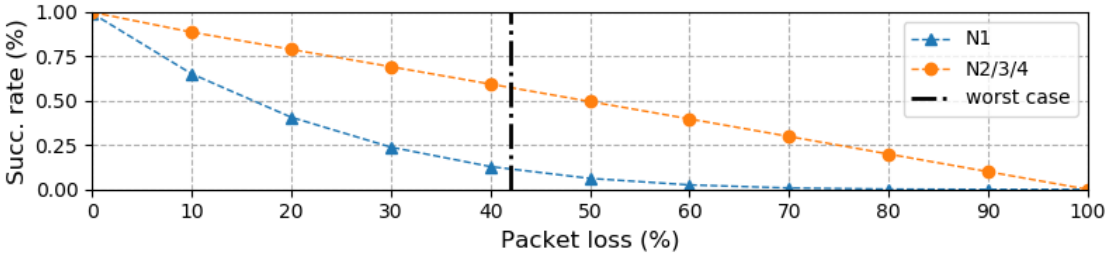


Figure 3.7: The success rate of *N1-4* under packet loss.

To have a deeper understanding how the network factor will affect the success rate, we leverage packet loss to demonstrate the capability of PMC. Figure 3.7 show that the success rate of *N1* decays much more than the other three attacks, because the attacker of *N1* needs to successfully send at least 4 malicious packets to ensure success. As *N2-4* target the same attack time window, they have the same success rate. For *N1-4*, the attacker should immediately launch the attack once the victim vehicle enters her communication range. Bai et al. [18] show that the packet loss rate (PLR) and the distance between two CV devices are positively correlated in real-world settings. In a freeway environment, the PLR is around 42% if two CV devices are $450m$ apart, in which $450m$ is the longest communication distance presented in their study. Thus, we highlight the success rate when the victim vehicle enters the attacker's communication range, which is the worst case for the attacker (PLR: 42%). Although the packet loss decreases the attack success rate, it also affects the transmission of normal packets, leading to the loss of critical CV safety packets.

Besides, CV communication is time-sensitive [68, 3, 4], so we would like to know the *time delay* caused by one round of *N1-4*, which is defined as the time duration from waiting for the trigger SPDU to successfully processing an SPDU from other vehicles. By knowing this, we can infer how long the CV network will recover from the attack if the attacker terminates attacking.

Table 3.4 shows that three of them can at least double the time delay in packet processing. During the experiments, we notice that there still exists *280 ms* time delay even if we disable the attacker, which is one-time delay introduced by P2PCD itself. For *N2*, the extra time delay introduced by the attacker is *90 ms*, around one broadcast interval, because the malicious learning response cancels out the learning request process triggered by the SPDU from `V1`. `V2` thus needs to wait for next SPDU from `V1`, which takes one more round of broadcast interval. For *N1*, *N3*, and *N4*, the extra time delay caused by the attack is about *300 ms*. If the attacker stops attacking at some time, it takes around three broadcast intervals (i.e., 300 ms) for `V2` to recover from DoS.

In *N1*, the extra time delay comes from the long processing time of P2PCD, due to the long time interval of the response backoff timer, with a random timeout value between 0 and 250 *ms*. As shown in Figure 3.5, the attacker sends malicious learning responses to `V1` right after `V1`

initializing the response backoff timer. Since the attack occurs at a very late stage, all the time before the transmission of the learning request become useless. Also, a new P2PCD learning process to the unknown certificate `ca1` will not be initialized again until both the response backoff timer of `V1` and the request active timer of `V2` expire. After that, `V2` needs to initialize P2PCD again; thus, one round of *N1* double the one-time delay of P2PCD. In *N3* and *N4*, `V2` is unable to process incoming trigger SPDUs until the request active timer expires. However, this timer is usually set to 250 ms, which largely increase the time delay.

3.4.2 PMP Vulnerabilities

CVANALYZER identifies 15 attacks in the PMPs of VENTOS [217] and PLEXE [174] (see Table 3.3). Among identified vulnerabilities, *A1-4* are not directly related to availability issues but are building blocks of other attacks. Although the PMPs analyzed are academic prototypes, our main contribution is the verification methodology, which can be generally applied to future PMP protocols. Our results demonstrate the necessity of such a systematic verification methodology: using manual efforts, a very recent work [1] can only uncover 1 vulnerability (A1). In contrast, using CVANALYZER for the same PMP implementation, we are able to automatically uncover not only the same one but also 14 more (A1-15), which demonstrates both substantially improved efficiency and effectiveness.

In the following descriptions, `V1` and `V2` still stand for vehicles. `V1` is a platoon leader, and `V2` is usually a follower. Their relative positions differ case by case.

3.4.2.1 PMP Attack Prerequisites

A1 and *A2* allow the attacker to become a valid platoon leader and follower. Abdo et al. [1] have demonstrated that *A1* can lead to the traffic collision and slow down the emergency vehicle. Although they do not directly cause security or safety breaches, we list *A1* and *A2* alone because they are prerequisites of other attacks. As described in §2.1.2, a platoon leader will send a merge request to a front platoon, if the combined platoon size is no greater than the optimal platoon size.

Thus, the attacker can claim herself as a front platoon to take over another platoon or initiate a merge maneuver to join a platoon, leading to the success of *A1* or *A2* respectively.

3.4.2.2 Split Trigger Attacks

Both *A3* and *A4* (see Appendix A for details on *A4*) can trigger the split maneuver at any positions. Without sacrificing her own speed stability, in *A3*, the attacker can further lead to a high-rate of vehicles entering and exiting a platoon, which decreases efficiency and safety [8].

Attack steps. In *A3*, the attacker first merges with `V1` as a malicious follower. Then, `V2` sends a `MERGE_REQ` to `V1` and join the platoon. At this time, the attacker intentionally sends a `LEAVE_REQ` with a wrong depth number of 2 to `V1`, in which the depth number indicates the splitting vehicle is `V2`. `V1` thus wrongly initiates the split maneuver at the position of `V2`. After the split process, `V2` receives beacon messages from the attacker and merges with the front platoon again, as described in §2.1.2. By repeatedly triggering merge and split maneuver of `V2`, the attacker downgrades the speed stability of `V2`.

Discussion. The reason for *A3* is that the platoon leader does not verify whether the platoon depth in the `LEAVE_REQ` matches with the sender ID or not. Usually, if the sender ID is related to unique signing certificates [91], it is difficult for the attacker to falsify the identity. However, the design of PMP uses the depth information as the identity, which can be easily modified by the attacker. Thus, PMP opens a door for the attacker to trigger the leave maneuver, leading to a split maneuver at arbitrary positions.

3.4.2.3 PMP Block Attacks

This is the most common type of vulnerabilities (*A5-14*) in the current PMP design of both VENTOS and PLEXE, which misleads the victim vehicle to stay at a busy state. We only describe *A7* here. Please refer to Appendix A for more details on others.

Attack steps. In *A7*, the attacker first joins the platoon by launching *A2* and aims at blocking the split maneuver. Usually, only the platoon leader can initiate the split maneuver, but the platoon

follower cannot. However, the attacker can leverage *A3* and *A4* to mislead the platoon leader to send a SPLIT_REQ to any specified platoon members. In *A7*, the attacker receives a SPLIT_REQ from V1 but chooses not to reply with a SPLIT_ACCEPT. Thereby, the platoon leader will keep waiting for the split reply. At this time, if V2, which is ahead of the attacker, approaches the destination and wants to leave the platoon, the leader V1 will not be able to process the leave request or manage the split process to create space for V2. Without enough space at the front and rear of the vehicle, it is dangerous for V2 to directly change the lane.

Discussion. The fundamental reason for *A5-14* is the lack of error recovery mechanism on communication failures. By design, the CV network stack does not provide reliable communication; it is the applications' responsibility to handle communication failures [93]. Researchers have already discussed the impact of communication failures on the CACC controller [8, 137], but do not pay much attention to communication failures on PMP. Also, we observe that PMPs in both VENTOS and PLEXE do not consider "offline" platoon members; thus, they do not design any error recovery mechanisms to reset the vehicle's state. Although we understand the PMPs of VENTOS and PLEXE are research prototypes, identified *PMP block attacks* still emphasize the importance of error recovery mechanisms in CV application design.

3.4.2.4 Inconsistency Attack

This attack aims at assigning a wrong depth number to a victim follower, which is inconsistent with the index in the platoon member list. The platoon depth is used in the split maneuver, so the inconsistent depth number can lead to failures of the split maneuver and the leader/follower leave maneuver.

Attack steps. In this attack, the attacker first joins V1's platoon as a follower. Then, the attacker slows down to create large gap (e.g., 100 m) between herself and V1. At this time, V2 change its lane and drives behind V1. V2 receives the beacon message from V1 and sends a MERGE_REQ to V1. After merging with V1, V1 updates its local state by appending V2's ID to the platoon member list, indicating the real platoon depth of V2 is 2. However, V2 only receives a beacon

message with the depth of 0 from the front vehicle V1; V2 thus wrongly sets its platoon depth to 1. At this time, the attacker sends a LEAVE_REQ to V1. Since, V1 thinks that the attacker is a middle follower, and V2 is behind the attacker, it sends a SPLIT_REQ to V2 to create rear space for the attacker. In VENTOS, we observe that CHANGE_PL does not present the absolute depth but carries the relative change of depth information, because it is convenient for the platoon leader to send all followers one CHANGE_PL rather than multiple different CHANGE_PL. During the split maneuver, V2 receives a CHANGE_PL from V1 with the depth change of -2. While updating the depth information locally, PMP of V2 throws an error for the invalid new depth: $1 - 2 = -1$, which may compromise the availability of PMP, as well as terminates the split maneuver.

Discussion. The reason for *A15* can be attributed to the inconsistent platoon view on the platoon leader and follower. When joining a platoon, the vehicle relies on the depth information in the beacon message from the front vehicle to set its own depth number, while the platoon leader simply appends a new member to the platoon member list without checking the relative location information. If the front vehicle is a benign last follower, no inconsistency will appear; otherwise, any CHANGE_PL from the leader to the victim vehicle will lead to a wrong new depth number. However, the attacker can either create a large gap for the victim vehicle (*A11*), or can send a beacon message with a wrong depth number if the attacker is the last follower.

3.5 Evaluation

In this section, we conduct extensive experiments and answer the following three research questions:

- **RQ1**: Are identified vulnerabilities practical in a real-world setting?
- **RQ2**: What are the security/safety impact of identified vulnerabilities?
- **RQ3**: What is the runtime performance of CVANALYZER?

3.5.1 RQ1: Practicality of Identified Attacks

We implement and validate all attacks from both P2PCD and PMP, detected by CVANALYZER, in a real-world testbed, which thus concretely demonstrates the effectiveness of CVANALYZER. Interestingly, we also find some poor implementation details in real-world CV devices that actually make our attacks easier.

3.5.1.1 Testbed Setup and Tool Preparation

As shown in Figure 3.8, we set up a CV network using three Cohda OBUs [45] in our lab. Among these three OBUs, denoted as OBU 1, 2, and 3 respectively, OBU 1 and 2 are used as victim CV devices, and OBU 3 is used as the attack device. To control the experiments, we connect a laptop with three OBUs via Ethernet connections.

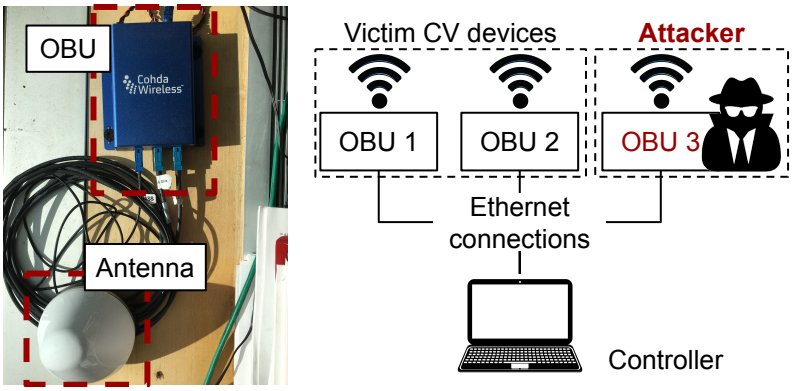


Figure 3.8: Testbed setup for attack validation.

The Cohda OBU that we use in our experiments is an ARM embedded device running Ubuntu 16.04. It implements the latest version of the CV network stack, which conforms with IEEE 802.11p [96, 97], IEEE 1609-2016 [91, 93, 92], and SAE J2735-2016 [46]. Notably, the implementation of IEEE 1609.2, called `Aerolink`, is developed by OnBoard Security [158] and closed source. As far as we know, `Aerolink` is the industry-leading implementation of IEEE 1609.2 and has been used by many manufacturers of CV devices (e.g., Cohda [191], Savari [159]) in the USDOT sponsored Connected Vehicle Safety Pilot program [158].

Victim OBU setup. To implement the CV communication model, both victim OBUs run a simple program that periodically broadcasts a correctly-signed SPDU. This broadcast-based communication also allows the attacker to observe all network traffic. For P2PCD, we place random data in the SPDU. For PMP, the SPDU stands for the beacon message, which contains the platoon ID and depth. Besides, both OBUs run PMP programs that are extracted from the source codes of VENTOS [217] and PLEXE [174].

For two different protocols (i.e., P2PCD, PMP), we assign different roles to OBU 1 and 2. In P2PCD attacks (i.e., *N1-4*), following the same assumption in §3.4.1, OBU 2 cannot verify packets sent by OBU 1 due to a missing certificate, so OBU 2 wants to initialize P2PCD to learn the unknown certificate from OBU 1. In PMP attacks (i.e., *A1-15*), by default, OBU 1 and 2 belong to the same platoon. OBU 1 and 2 are the platoon leader and the platoon follower respectively.

Tool preparation. To launch the attacks, we need to prepare tools that allow us to (1) parse and construct arbitrary packets and certificates, and (2) sign and verify CV network packets correctly. For (1), we use `asn1c` [219] to extract C data structures used by CV network services from ASN.1 modules in protocol specifications, and port platoon message types from the source codes of VENTOS and PLEXE. For (2), we follow IEEE 1609.2 to implement the signing and verification functionalities. We start from ECDSA APIs provided by `OpenSSL 1.1.0j` [162]. The elliptic curve and the hash function that we use with ECDSA is `NIST P-256` and `SHA-256`, respectively. We cross-validate the correctness of our tools using APIs of Cohda CV network stack. The Cohda CV network stack can process packets and certificates generated by our tool without throwing any errors.

Certificate configurations. As *N1-4* require triggering P2PCD, we need to configure the preinstalled certificates in both victim OBUs to ensure that OBU 2 cannot construct a certificate chain while verifying packets sent by OBU 1. Both OBU 1 and 2 can correctly verify packets from OBU 3 (attacker). First, we use our certificate generator to construct a Root CV certificate, referred as `root`, which is trusted by all three OBUs. Then, we use `root` to issue two intermediate Certificate Authority (CA) certificates: `ca1` and `ca2`. We add both `ca1` and `ca2` to the local

certificate database of OBU 1, but only add `ca2` to the database of OBU 2. To generate end-entity certificates for signing packets, we utilize `ieeeAcfGenerator` in Cohda SDK to issue two batches of certificates: `batch1` for OBU 1 and `batch2` for OBU 2. Each batch is an `Aerolink`-specific file and contains 20 end-entities certificates. Besides, we use `ca2` to issue another end-entity certificate `ee3` for the attacker so that OBU 1&2 can construct a valid certificate chain for packets sent by the attacker.

Apart from generating these normal certificates, we also need to construct certificates that can cause hash collisions. In *N1* and *N2*, the first certificate in the malicious learning response should match with the low-order 3-byte and 8-byte hash value of the unknown certificate respectively. We therefore use our certificate generator to construct two CA certificates: `ca1-h3` and `ca1-h8`, which can lead to 3-byte and 8-byte hash collision with `ca1`.

Attack programs. Following the attack processes in §3.4, we implement different attack programs. For each attack program, we set the start condition and the fail condition. The attack programs will stop only if the fail conditions are satisfied; otherwise, they will keep running. For P2PCD attacks, the attack fails if she observes any learning response from OBU 1. For example, the attack program for *N1* will send malicious learning responses after observing a learning request sent by OBU 2 (i.e., `Vehicle 2` in Figure 3.5). If it observes a learning response sent by OBU 1, the program will stop, which means that the attack fails. For PMP attacks, the attack fails if the victim platoon member can still finish the merge, split, leave, or dissolve maneuver.

3.5.1.2 Validation Results

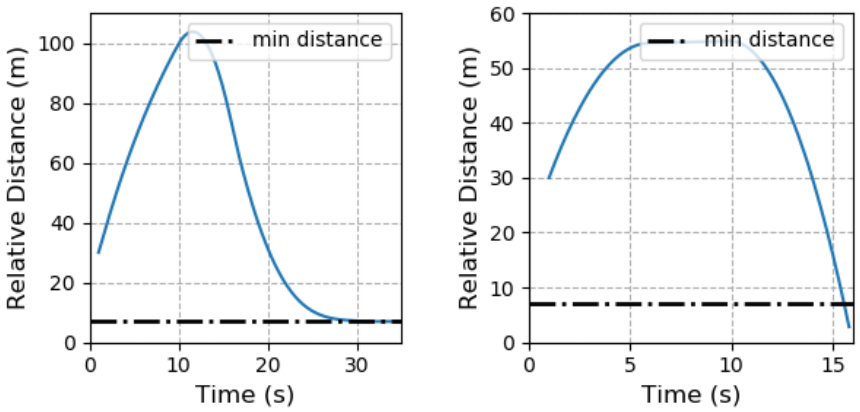
In the real-world experiments, we find that all attacks from P2PCD and PMP are successfully validated. Interestingly, we further find that some implementation details in `Aerolink` can actually make P2PCD attacks, *N1* and *N2*, even easier and even block the CV communication indefinitely.

First, we observe that *N1* and *N2* can indefinitely block the P2PCD learning process. Based on our model-checking findings in §3.4.1, once the adversary stops sending malicious learning responses, the victim devices should eventually be able to recover from DoS. However, in our

real-world experiments, we find that even after the attack program terminates, OBU 2 still cannot learn the correct unknown certificate from OBU 1. After analyzing the execution log, we find that OBU 1 keeps sending the fake certificate (i.e., `ca1-h3`), while OBU 2 sends learning requests for the unknown certificate `ca1` to OBU 1. By design, a CV device responds to an incoming learning request only if the learning request field matches with a signing certificate which is recently used by that device. With the help of a binary disassembler called `Hopper` [23], we find that `Aerolink` actually does not check whether the certificate used for a learning response is indeed a recently used certificate. For example, in *N1*, OBU 1 stores the fake certificate (i.e., `ca1-h3`) carried by the malicious learning response from the attacker. Thus, during the preparation of the future learning response, OBU 1 has two candidates, `ca1` and `ca1-h3`, as they have the same low-order three-byte hash. When receiving learning requests, OBU 1 always picks `ca1-h3` and sends it to OBU 2, which thus permanently prevents OBU 2 to learn the correct certificate.

Second, to launch *N1*, we find that the attacker only needs to send 3 malicious learning responses instead of 4. Before running real-world experiments, we first measure the response threshold in `Aerolink`, and find that the threshold set in `Aerolink` is actually 2 instead of 3 in the protocol specifications. This finding is also confirmed using `Hopper`. In this case, the attacker only needs to send 3 malicious responses to succeed. Although this may not be a big improvement for the attacker, it still uncovers an implementation choice in `Aerolink` that is unexpectedly favorable to the attacker.

Third, we find that *N2* only requires 3-byte hash collision rather 8-byte hash collision, which largely lowers the bar of launching *N2*. In P2PCD, by design, a CV device records an unknown certificate by adding the identity of that certificate (i.e., an 8-byte hash value) into a queue. If the 8-byte hash of a certificate in an incoming learning response matches with any entries in the queue, that entry will be removed. To launch *N2*, the attacker has to intentionally cause the 8-byte hash collision to let the victim CV device wrongly remove an entry in the queue. However, according to our binary analysis through `Hopper`, we find that `Aerolink` actually uses a 3-byte hash of the unknown certificate to record its status. Therefore, in our real-world experiments, we use



(a) No collision, FCW (b) Collision

Figure 3.9: Relative distance between the leading vehicle (`V1`) and the following vehicle (`V2`).

`ca1-h3` in *N2*, and the results further validates this finding. Later in §3.6, we will show why this small truncated hash (e.g., 3-byte hash) is not secure enough. Although the protocol specification does not clearly state how to record unknown certificates, Annex D in IEEE 1609.2 [91] gives an example of P2PCD implementation that uses the 8-byte hash as the identity to record the unknown certificate. Also, while recording the unknown certificate, the most complete identity about the unknown certificate is the 8-byte hash value. A CV network implementation should always use complete information rather than truncated information.

3.5.2 RQ2: Attack Impact

The following two case studies demonstrate the impact of identified attacks: (1) P2PCD attacks can lead to traffic accidents, which eliminates the benefits of V2V safety applications (e.g., Forward Collision Warning (FCW)); (2) PMP attacks can affect the speed stability of the victim vehicle.

Simulator setup. To evaluate the impact of identified attacks, we use a simulator, VENTOS (VEhicular NeTwork Open Simulator) [217], so that we can demonstrate the driving behavior under attacks. VENTOS is built upon SUMO road traffic simulator [203] and OMNeT++ [157]/Veins [215, 200] network simulator. These simulators [203, 157, 215] have been widely used in academia, industry, and the government. We configure it to use the models for the

IEEE 802.11p [96] protocol for CV communication. Based on our reverse engineering and study on `Aerolink` (§ 3.5.1), we port the digital signature and P2PCD in IEEE 1609.2 to the simulator to secure BSMs and PMP commands. All secured packets are then transmitted through Wave Short Message Protocol (WSMP) and are directly sent to the data-link layer which uses continuous channel access based on IEEE 1609.4 [92].

Vehicles	Initial Speed	Max. Speed	Max. Decel.	Length
Leader (`V1`)	30 m/s	30 m/s	5 m/s^2	10 m
Follower (`V2`)	20 m/s	30 m/s	2 m/s^2	5 m

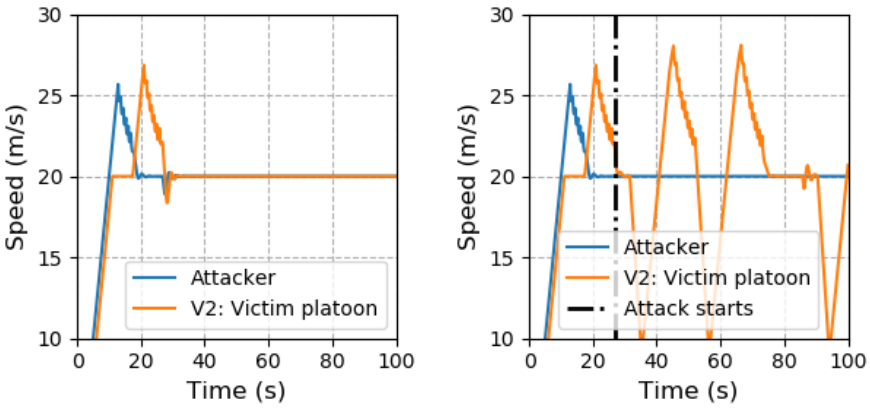
Table 3.5: Vehicle parameters in the rear-end collision scenario.

3.5.2.1 Safety Impact

By design, the CV safety application promises to increase personal safety [210]. However, our experiment results show that P2PCD attacks can fully eliminate the benefits of CV applications (e.g., Forward Collision Warning (FCW)), violating the original goal of CV applications.

Rear-end collision scenario w/ FCW. We first set up a rear-end collision scenario and demonstrate that vehicles with Forward Collision Warning (FCW), a V2V safety application, can avoid the accident (Figure 3.9a). The rear-end collision scenario includes a leading vehicle (`V1`) and a following vehicle (`V2`) with the initial parameters in Table 3.5. FCW alerts the driver in order to help avoid the severity of crashes into the rear end of other vehicles on the road [210]. We follow the FCW's design in Cohda SDK to actively monitor the distance between two vehicles. Once the distance is smaller than the safe distance, FCW will warn the driver. As FCW does not directly control the vehicle, after receiving FCW warnings, we ask the simulated vehicle to maintain a safe speed. Notably, we leverage Krauss car-following model [109], which is collision-free, to calculate the safe distance and safe speed.

During the simulation, both vehicles drive in the same lane. By exchanging BSMs, they can monitor each other's speed, position, and acceleration. The initial distance between two vehicles is 30 m, which is smaller than the safe distance at that time, thus triggering FCW. After starting



(a) Before attack (b) After attack

Figure 3.10: Speed profiles in *A3* (split trigger attack).

the simulation for 10 s, V1 suddenly stops at the maximum deceleration (i.e., 5 m/s^2). Figure 3.9a shows that, before 10 s, V2 keeps increasing the distance to the leading vehicle due to the FCW. Therefore, after the leading vehicle suddenly decelerates, V2 has enough space to slow down safely.

Vehicles w/ FCW under attacks. Then, we place an attacker on the roadside who follows § 3.4.1 to launch P2PCD attacks and aims at causing traffic accidents, leading to a rear-end collision shown in Figure 3.9b. At the beginning of the simulation, both vehicles launch P2PCD to exchange certificates so that they can verify and process following BSMs. However, P2PCD attacks prevent them from learning certificates, meaning that they cannot process any BSMs from the peer vehicle. During the simulation, we observe that FCW is never triggered, so V2 accelerates to the maximum speed and follow V1. At 10 s, V1 starts decelerating at the maximum deceleration (i.e., 5 m/s^2). Since two vehicles are too close to each other (i.e., 54 m), and the maximum deceleration of the V2 is 2 m/s^2, V2 eventually collides into the rear end of V1.

3.5.2.2 Traffic Efficiency Impact

By design, CACC aims at increasing traffic throughput and improve traffic flow stability [223, 175, 136]. However, *A3* and *A4* can interfere with the traffic flow stability, even without sacrificing

her own speed stability, which violates the design goals of CACC. We place V1, the attacker, and V2 sequentially in the same lane and follow the attack steps of *A3* to run the simulation for 100 seconds. Figure 3.10 presents the speed profiles of V2, the victim platoon. In the normal case (Figure 3.10a), all vehicles will eventually reach a stable speed of 20 m/s; after launching the attack starting around time 27 seconds, we increase the standard deviation of V2's speed by 43%, further disturbing the following traffic.

3.5.3 RQ3: Performance of CVANALYZER

Table 3.6 presents the runtime performance of CVANALYZER. We run CVANALYZER on a server with four 2.60GHz (8-core) CPUs and 128G memory. CVANALYZER first explores all reachable states and then verifies given properties. Notably, without applying the state reduction, these two model checking tasks will take too long to explore reachable states. The results highlight the importance and effectiveness of state reduction.

Attacks	Distinct States	Model Checking Duration
P2PCD	2209351	16s
PMP	142133161	1h 35min

Table 3.6: Runtime statistics of CVANALYZER.

3.6 Defense Proposals

Based on the discussions from previous sections, we propose defense solutions at the protocol design level:

1. Mandate verification for all learning responses;
2. Increase the truncated hash size for the issuer field in the certificate and the learning request field in SPDU;
3. Disallow unicast learning requests;

4. Bind the sender identity with the CV certificate;

5. Track platoon configuration data locally or remotely;

6. Design and integrate an error recovery mechanism.

Defense against N1 and N2. Solution 1 and 2 are proposed for *N1* and *N2*. Solution 1 by nature prevents *N1* and *N2* with local certificates. However, such solution can be evaded if attackers are still able to collect legitimately signed certificates with the attacker-desired hash values by sniffing CV network traffic. As estimated in Table 3.7, as long as the attacker can collect over 12000 different certificates, she can almost guarantee (more than 98% probability) that she can always have a certificate ready for triggering a 3-byte hash collision, which thus allow her to still launch *N1* and *N2* in real time. Collecting this many different certificates is completely realistic, considering that such collection process can be done offline. In addition, the collection process can also be greatly accelerated since the attacker can actively broadcast learning requests to trigger surrounding vehicles to return certificates with desired hash values, and also can place multiple attack devices in different locations to parallelize the collection process.

Number of hash values (k)					
Prob. of hash collision (p)	Number of bits of the hash value (n) 24	64	80	256	512
0.5	4823	5.06^{9}	1.29^{12}	4.01^{38}	1.36^{77}
0.99	12431	1.30^{10}	3.34^{12}	1.03^{39}	3.51^{77}

Table 3.7: Number of hash values needed for hash values of n-bits to cause a hash collision probability at p.

Solution 2 aims at increasing the difficulty of causing a hash collision, the key enabler for *N1* and *N2*. As shown in Table 3.7, it will be much more difficult for the attacker either to compute or to gather proper malicious learning responses. However, this will increase the DSRC packet size and thus may decrease the network performance, e.g., increasing network latency. We have reached out to the protocol developer, and confirmed that it is indeed a design choice to reduce the

DSRC packet size. Thus, when applying Solution 2, the new size of the truncated hash type needs to be carefully chosen to balance such trade-off between security and protocol performance.

From our discussion above, neither solution 1 or 2 can fully eliminate the attack possibilities for *N1* and *N2*. Thus, to maximize the chance of preventing the attack in practice, the best choice would be using them jointly.

Defense against N3 and N4. Solution 3 is proposed for *N3* and *N4*, which thwarts both attacks by making it impossible to unicast the malicious learning request to block the P2PCD process. However, the down side is that this may break designed usage of unicast-based learning request. For example, as specified in IEEE 1609.0-2019 [95], CV applications will decide whether to use either unicast or broadcast, while receiving advertised services. Systematically understanding this trade off requires surveying and quantifying the demands of unicast-based learning requests at the CV application level, which we leave as future work.

Defense against A3. Solution 4 can prevent the attacker from triggering the split maneuver at arbitrary positions, but cannot stop her splitting succeeding platoon members. The certificate defined in IEEE 1609.2 [91, 29] provides a unique identity for each CV device. Safety-critical CV applications like PMP should always use unique and secure identities (e.g., certificates) rather than using self-defined identity (e.g., depth number), which is easily spoofed by the attacker. However, the attacker can still send a `LEAVE_REQ` to split at the succeeding vehicle and herself, which is a designed follower-leave behavior. The attacker can then join back to the platoon and launch the attack repeatedly. To completely address *A3*, we may require the assistant of misbehavior detection [29]. For example, a vehicle that keeps leaving and joining a platoon is highly suspicious. Designing an effective misbehavior detection requires comprehensively characterizing malicious behaviors, which we leave as future work.

Defense against A4 and A15. Solution 5 aims at eliminating wrong and inconsistent platoon information caused by *A4* and *A15*. In centralized PMP, a platoon leader is responsible for passing platoon configuration data to the new leader, when it leaves the platoon. The new leader can only accept the information from the old leader because it does not store any platoon configuration

data. The design goal of the centralized PMP is to improve coordination efficiency and to enhance privacy because followers dynamically enter and exit the platoon [8]. However, the centralized design sacrifices the security, as a malicious leader can provide wrong platoon configuration data. To address *A4* and *A15*, on one hand, the platoon members can maintain a local copy of platoon configurations. On the other hand, RSUs can also provide services to remotely assist platoon members for tracking platoon configurations and guarding PMP commands [1]. As RSUs are often deployed and managed by trustworthy authorities, platoon members can rely on the infrastructures to correct wrong or inconsistent information.

Defense against A5-14. Solution 6 is straightforward and proposed for all PMP block attacks. As we mentioned before, CV applications should design their own error recovery mechanisms. With the error recovery mechanism, PMP should be able to recover from continuous packet loss. For example, PMP can define the retransmission and timeout threshold to avoid hanging at specific states. Apart from the classic solution to communication failures, it's worthing noting that PMP should also adjust the intra-platoon spacing between the "offline" member and the trailing platoon members accordingly to avoid traffic collision. If necessary, the platoon leader can dissolve the platoon and falls back to ACC mode.

3.7 Conclusion

In this chapter, we presents CVANALYZER that harnesses the attack discovery capability of the general model checker and the quantitative threat assessment of the probabilistic model checker to automate the analysis. CVANALYZER successfully detects 4 new DoS attacks in P2PCD and 15 attacks in PMP; also, we construct practical exploits and validate them in a real-world testbed. We have reported 4 P2PCD attacks to IEEE 1609 Working Group [94] and received confirmations. Also, we discuss the fundamental reasons for these vulnerabilities and propose effective mitigation solutions.

Future work. In the future, we would like to extend CVANALYZER to verify more secu-

rity properties, such as unlinkability. Although we only inspect the availability property in this dissertation, CVANALYZER is actually general and can be extended to improve the verification capabilities. On the other hand, CVANALYZER can be also extended to support other protocols in the context of CV (e.g., SCMS [29]). Also, we would like to improve the usability of CVANALYZER. For example, we can introduce an intermediate representation for the model that can be automatically converted into the modeling language used by different model checkers. Therefore, we do not need to write the model twice for two different model checkers.

CHAPTER 4

Practical Broadcast Authentication Approach for the Next-Generation In-Vehicle Network

4.1 Introduction

Due to the increasing data transfer needs of diverse sensors (e.g., cameras, LiDAR) in modern vehicles, existing in-vehicle networks (e.g., CAN, FlexRay, MOST) cannot meet the critical high-bandwidth, low-latency, and real-time network requirements. For example, the maximum bandwidth of CAN and FlexRay are 1 Mbps and 10 Mbps, respectively. To this end, Automotive Ethernet [124, 100, 85, 89] is considered to be the next-generation in-vehicle network, because of its high bandwidth, high throughput, and low cost characteristics. Nowadays, many car manufacturers are planning to move to Ethernet for all classes of cars. Hyundai uses Automotive Ethernet for infotainment systems in upcoming cars. Also, Volkswagen adopts Automotive Ethernet for driver-assist systems [100].

While the Automotive Ethernet can greatly benefit in-vehicle communication, no common standard has been established for the security protocol of Automotive Ethernet. Security is especially important as it is conceivable that an electronic control unit (ECU) can be compromised, subsequently violating the integrity of any unencrypted communication among ECUs. This could lead to sophisticated attacks such as remote braking, as some of the components in the car allow wireless network access [34, 108].

To better understand the security protocol candidates, we analyze three candidate protocols:

MACsec, IPsec, and TLS. We summarize a comprehensive list of security and performance requirements for securing in-vehicle communication. Our goal is to ensure that the in-vehicle security protocol can secure in-vehicle communication and does not incur high performance overhead. Our analysis results indicate that three candidate protocols cannot fully satisfy all identified requirements. Most importantly, they can only provide source authentication for unicast communication rather than multicast/broadcast [1] communication, which is crucial for secure in-vehicle communication. If an attacker participates in a broadcast group, a single malicious packet can potentially impact multiple receivers. To prevent such malicious behavior, the receiver must be able to verify the identity of the sender. Timed Efficient Stream Loss-Tolerant Authentication (TESLA) [167] is a well-known protocol to ensure source authentication. Nonetheless, TESLA introduces unnecessary delay due to the time-delayed key disclosure, which is also confirmed in our evaluation (§4.6.5).

Furthermore, the three candidate protocols do not consider the Denial-of-Service (DoS) prevention [7] in their designs, in the form of attackers sending a large volume of packets. Note that DoS prevention is essential for guaranteeing in-vehicle network security. For the common CAN injection attack, sending malicious CAN frames at a high frequency can "mute" the legitimate ECU and force others to consume forged CAN frames [39]. On the other hand, deploying a security protocol also opens a door for new DoS attacks. For example, Kim et al. [106] demonstrate that many IoT protocols are vulnerable to DoS attacks, caused by heavy cryptography operations (e.g., signature generation). Without addressing these challenges, the in-vehicle network is still vulnerable to cyberattacks, susceptible to safety accidents.

To address the issues mentioned above, we propose GATEKEEPER, a gateway-based broadcast authentication protocol. It uses symmetric cryptography as the building block because symmetric cryptography is much more efficient than asymmetric cryptography. Also, ECUs often have limited resources and cannot afford the expensive asymmetric cryptography operations. Apart from that, GATEKEEPER takes advantage of the centralized topology of the in-vehicle network (§ 4.2.1).

[1] Multicast and broadcast are used interchangeably for convenience.

We introduce an on-path authenticator, which co-locates with the centralized gateway or domain controllers. By doing so, we can benefit from the rich computation resources provided by the gateway or domain controllers. The on-path authenticator can help receivers verify the sender's identity during the transmission and thus avoid unnecessary network forwarding. Additionally, based on the time-lock puzzle [185], we design a DoS protection approach that can slow down unexpected high-throughput traffic from the attacker and only introduce minimal overhead at the authenticator side.

To summarize, we make the following contributions:

- We conduct a systematic analysis of MACsec, IPsec, and TLS for in-vehicle Ethernet network, covering security and performance requirements, which shows that *source authentication* and *DoS prevention* are two missing but essential security properties for these candidates.

- We propose a novel gateway-based broadcast authentication protocol, GATEKEEPER, to ensure source authentication for in-vehicle Ethernet network. In addition, we integrate a DoS protection approach, which is based on the time-lock puzzle [185], to alleviate the impact of an aggressive attacker who aims at frequently triggering computationally heavy operations at the authenticator. In addition, we formally verify that GATEKEEPER achieves the desired security properties, using the Tamarin prover [142], which further strengthens the security guarantee of GATEKEEPER.

- We build a Docker-based testbed and use a *realistic* ECU board to calibrate its performance for our evaluations. Based on the testbed, we prototype GATEKEEPER and compare its performance with the TESLA protocol. The evaluation results show that GATEKEEPER incurs low latency overhead (e.g., 0.03 ms latency overhead for CAN) and significantly outperforms TESLA on both CAN and LiDAR transmission scenarios. Table 2.1 further highlights the efficiency of the design of GATEKEEPER, compared to other existing works [107, 133, 64].

4.2 Network Topology

In-vehicle network architectures vary considerably in the manufacturer, type, or even configurations [232]. Since no generic architecture exists, in this section, we follow existing works [232, 67] and assume a general architecture with different domains to motivate the security requirements in §4.4.

4.2.1 In-vehicle Ethernet Network Architecture

In this chapter, we assume that all connections in Figure 4.1 are realized with Automotive Ethernet. Automotive Ethernet, running at up to 100 Mbps, is favored for the next-generation in-vehicle network, compared with existing CAN, MOST, and FlexRay technologies. The adoption of Ethernet also changes the network topology from a bus system to a star system with switches [232, 140].

As shown in Figure 4.1, there exist different domains in the in-vehicle network, such as powertrain, chassis, body, infotainment, and ADAS (advanced driver-assistance systems). For each domain, Electronic Control Units (ECUs) are connected to the domain controller, which has Layer 2 (L2) switching capabilities. All domains are isolated from each other and can be further configured to form different security zones through VLANs [100]. Besides, all domain controllers are then connected to the gateway. Note that the gateway should have Layer 3 (L3) routing capabilities to support inter-domain communication. Note that, the infotainment domain often offers various interfaces for *external* communication [7], such as Bluetooth, USB, Wi-Fi, and V2X (vehicle-to-everything), which is out of the scope. Thus, we focus on *in-vehicle communication* over Ethernet network in the following analysis in §4.4.

4.2.2 Communication Patterns

There exist different communication patterns for the Automotive Ethernet and the in-vehicle communication [140].

Unicast communication is the core transmission mode of a switched Ethernet network. Mes-

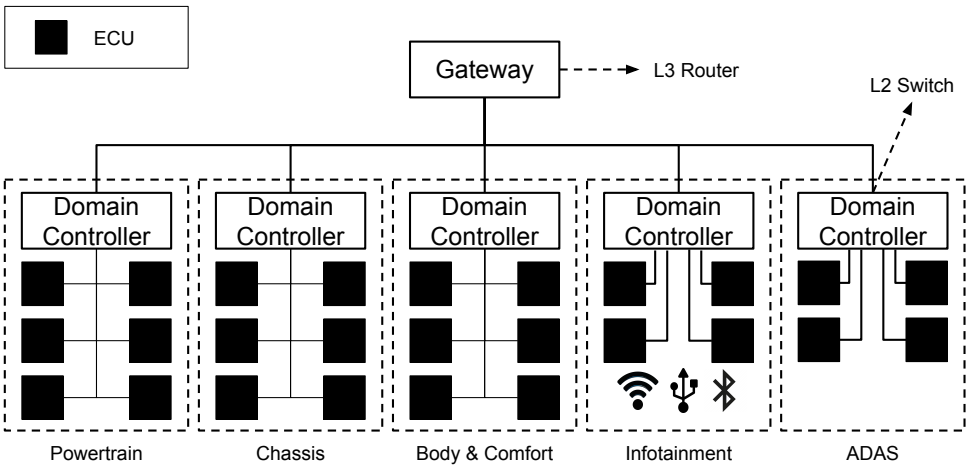


Figure 4.1: Ethernet-based in-vehicle network architecture.

sages that are sent to a single network destination are often transmitted through unicast.

Broadcast communication is one of the most commonly used communication patterns for in-vehicle communication nowadays. ECUs are connected via a bus system and thus can receive all data that is available on the bus. For compatibility, we can assume that ECUs still broadcast control messages in the in-vehicle Ethernet.

Inter-domain communication delivers messages across domains, which is more and more common with the development of the autonomous vehicle. For example, the head unit in the infotainment domain may receive the picture of a rear-view camera from the ADAS domain.

4.3 Threat Model

In this chapter, we consider both external and internal attackers. As shown in previous work [171, 34, 108], to launch an automotive attack, the attacker always aims at gaining access to the in-vehicle network (e.g., CAN bus) to inject CAN frames, resulting in various attack sequences (e.g., acceleration, hard brake). To achieve the in-vehicle network access, the attacker can either (1) attach/tap a physical device (e.g., OBD-II dongle [225]) to the in-vehicle network or (2) compromise an existing ECU remotely [144, 34]. The former is an external attacker, while the latter is an internal attacker. The difference between these two cases is that the external attacker cannot

access the key materials of the compromised ECU. In contrast, even if the in-vehicle network is secured by security protocols, the internal attacker can still participate in the communication and launch attacks (e.g., sending spoofed CAN frames).

Although both external and internal attackers are widely studied over the last decade, we highlight that it is not easy to compromise a safety-critical ECU. First, ECUs with remote interfaces (e.g., Bluetooth, Wi-Fi) often belong to a less safety-critical domain (e.g., infotainment), which unlikely affects safety-critical domains. Second, it is time-consuming to compromise an ECU and achieve in-vehicle network access [144, 225]. For example, in the Jeep Cherokee hack [144], the attacker needs to analyze and modify the ECU firmware to gain the in-vehicle network read/write permission.

In summary, we assume that the attacker has the following broad set of capabilities:

- The attacker can either attach her own device to the network or compromise an in-vehicle ECU [34, 108].
- The attacker owns the valid key materials and can participate in the in-vehicle communication. For example, the attacker can extract the key materials of the compromised ECU.
- The attacker can eavesdrop on the network traffic at her position and send arbitrary packets.
- The attacker cannot break existing cryptography algorithms, meaning that she cannot infer the secret key used by others.

Since the attacker can send arbitrary packets after accessing the in-vehicle network, the following attacks are feasible:

- *Replay attack*: the attacker records a packet then sends it again later in the hopes of it having the same effect that it had the first time. For example, the attacker captures a CAN frame that accelerates the vehicle and replays this frame to keep the vehicle accelerating.
- *Spoofing attack*: the attacker impersonates herself as another in-vehicle node by falsifying data (e.g., CAN frame), to gain an illegitimate advantage.

- *Denial-of-service (DoS)*: the attacker keeps sending packets in the network with the goal of resource exhaustion, which is also referred to as the flooding attack. This behavior is common for CAN injection attack [39], because the attacker needs to broadcast forged CAN frames at a *high frequency* to override legitimate CAN frames with the same ID. Another example is that the attacker may intentionally trigger computationally heavy operations (e.g., cryptography operations [106]) and tries to slow down or crash the victim.

4.4 Analysis of Security Protocols

In this section, we first summarize a list of security and performance requirements for secure in-vehicle communication. The requirement list is complete to our best knowledge and are derived from four different perspectives: (1) in-vehicle communication patterns (§4.2.2), (2) general security properties (i.e., integrity, confidentiality, and authentication), (3) potential security threats (§4.3). (4) in-vehicle performance challenges (Table 4.1). We then analyze three security protocol candidates: MACsec [86], IPsec [103, 104], and TLS [183], and discuss their limitations.

4.4.1 Requirement Items

Security requirements. We summarize 8 security requirements (SRs) as listed below. Our goal is to ensure that the in-vehicle security protocol can provide a secure channel for in-vehicle communication including three patterns mentioned in §4.2.2 and prevent potential security threats.

SR1 *Key Material*: Each in-vehicle node MUST own a cryptography key material, which will be used to protect the in-vehicle communication. Specifically, the security protocol MUST support both the symmetric key (e.g., pre-shared key) and the certificate (e.g., public/private key).

SR2 *Integrity*: The security protocol MUST support integrity protection to prevent tampering with any data sent over the communication channel.

SR3 *Confidentiality*: Confidentiality is an OPTIONAL requirement. It can prevent eavesdropping on the network data. It depends on the use scenarios if this requirement should be enforced

(e.g., the transmission of privacy-related data). Besides, enforcing confidentiality brings performance overhead, as shown in Figure 4.6. Therefore, we leave this as an optional requirement.

SR4 *Authenticity*: The security protocol MUST support authentication for the in-vehicle communication. As mentioned in § 4.3, the attacker is able to join the in-vehicle network. There is no default mechanism of verifying the authenticity of an added malicious device to the in-vehicle network. Therefore, the security protocol needs to ensure that only pre-authorized ECUs are allowed to participate the in-vehicle communication. Since both the symmetric key and certificate must be supported (SR1), the authentication can happen via a symmetric pre-shared key (PSK) or asymmetric cryptography (e.g., RSA, ECDSA).

SR5 *Source Authentication / Spoofing Attack Prevention*: For broadcast communication, the protocol participants within a group MUST be able to verify the identity of the sender; otherwise, a spoofing attack can occur within a given group, which is a well-known vulnerability for CAN bus.

SR6 *End-to-End Protection*: The security protocol MUST offer end-to-end security (e.g., integrity, optional confidentiality, and authentication), in which "end" means an ECU. If confidentiality is enabled, only the end nodes can decrypt the packet. Due to the integrity protection, any other nodes (e.g., domain controller, gateway) along a network path cannot modify the packets.

For the given network architecture in this chapter (§4.2.1), there are two types of end-to-end protection: (1) intra-domain end-to-end protection (security protocol at L2 is sufficient) and (2) inter-domain end-to-end protection (security protocol at L3 or above is sufficient).

SR7 *Replay Attack Prevention*: The security protocol MUST be able to prevent *replay attacks*. For example, the security protocol can attach a counter [118, 171, 17] or a timestamp in the packet. This will allow the receiver to discard any messages with a repeated/outdated counter or timestamp.

SR8 *DoS Prevention*: The security protocol MUST provide the DoS protection mechanism and prevent the attacker from aggressively consuming network/computation resources, as briefly discussed in § 4.1 and § 4.3.

Performance requirement. The primary performance requirement is "*efficient communica-*

Data Type	Max. Packet Size (byte)	Service Interval (ms)	Bandwidth	Max. Latency (ms)
Data 1	20	10-100	1.6-16 Kbps	0.1
Data 2	20	10-100	1.6-16 Kbps	10
Camera	786	0.25	25.1 Mbps	45
LiDAR [216]	1248	0.5	19.9 Mbps	45
Audio	1472	8.4	1.4 Mbps	150
Video	1472	1	11.8 Mbps	150

Table 4.1: Performance requirements of different types of traffic (adapted from [232, 119, 123]).

tion", because in-vehicle ECUs often have limited computational resources. For example, the CPU speed of the NXP MPC5748G development board is only 160 MHz [195], which is no more than 10% of a 2 GHz desktop CPU. Most importantly, the timely delivery of safety-critical data should be guaranteed. That is, the security protocol MUST NOT incur high overhead (e.g., time, bandwidth) that violates the performance requirements of different types of in-vehicle traffic (Table 4.1).

As shown in Table 4.1, different types of traffic have different performance requirements Except for LiDAR, other information are based on previous literature [232, 119, 123]. Both Data 1 and 2 are critical control data but are assigned with different latency (i.e., end-to-end delay) requirements. Camera is quite essential for ADAS and can help enhance driving safety, so we should ensure the transmission of Camera data should be finished in time. We further add LiDAR traffic characteristics into Table 4.1. LiDAR and Camera are complementary and are essential for the perception module of an autonomous vehicle; therefore, we assign the same latency requirement as Camera for LiDAR. Multimedia data, like audio and video, are often related to entertainment and thus have a less restricted latency requirement, 150 ms. In summary, critical control data (i.e., Data 1 and 2) has the strictest latency requirement ($\leq$ 10 ms). Sensor data (i.e., camera and LiDAR) are essential for the perception of an autonomous vehicle so that their transmission latency must satisfy the requirements from the perception module (45 ms). Multimedia (i.e., audio and video) is for entertainment and thus has a less restricted latency requirement (150 ms).

4.4.2 Comparison of Security Protocols

Based on the requirement list (§4.4.1), we compare existing security protocols: MACsec [86], IPsec [103, 104], and TLS [183]. For each security requirement, we refer to the protocol specifications to check whether the corresponding protocol satisfies the requirement. The quantitative evaluation of the performance requirement is presented in § 4.6.4.

ID	Requirements	MACsec [86]	IPsec [103, 104]	TLS 1.3 [183]
SR1	Key Material	✓	✓	✓
SR2	Integrity	✓	✓	✓
SR3	Confidentiality	✓	✓	✓
SR4	Mutual Authentication	✓	✓	✓
SR5	Source Authentication	✗	✗	✗
SR6	End-to-end Protection	Layer 2	Layer 3	Layer 4
SR7	Replay Prevention	✓	✓	✓
SR8	DoS Prevention	✗	✗	✗

Table 4.2: Comparison of MACsec, IPsec, and TLS.

As shown in Table 4.2, all three protocols can satisfy general security requirements, SR1-4, and replay attack prevention (SR7). For end-to-end protection, IPsec and TLS can secure both intra-domain and inter-domain communication. However, MACsec can only protect intra-domain communication, and thus is insufficient in our assumed architecture.

As mentioned in §4.2.2, broadcast is commonly used for in-vehicle communication. However, all three protocols are designed for two endpoints; thereby, they do not support the protection for broadcast communication. Moreover, TLS relies on TCP and thus does not support broadcast communication [2]. Indeed, with extra configurations [86, 63], MACsec and IPsec can secure broadcast communication if a group of nodes shares the same secret key, but they cannot ensure source authentication. Without fixing this problem, the attacker can still forge malicious packets and further break safety-critical functionalities.

For DoS attacks, none of these security protocols can completely prevent them. We can rely on other approaches to alleviate the impact of DoS. For instance, we can define filtering policies in

[2]DTLS may support broadcast protection [105, 134], but no standardized version has been finalized.

the domain controller and the gateway to prevent traffic flooding. An Intrusion Detection System (IDS) can also help detect and prevent malicious behaviors, such as constantly sending malicious packets to trigger heavy crypto operations in peer nodes.

4.5 Design of GATEKEEPER

In this section, we present GATEKEEPER to ensure source authentication and DoS protection in an in-vehicle Ethernet network. We first describe the high-level design in § 4.5.1. Detailed workflow of GATEKEEPER is presented in § 4.5.2. Then, we discuss the DoS threat against GATEKEEPER and propose a defense mechanism for it. At last, we show how we use Tamarin prover to verify the security properties of GATEKEEPER.

4.5.1 High-Level Design

As emphasized before, broadcast communication is commonly used in an in-vehicle network. Network participants (e.g., ECUs) form a communication group. A single packet can reach multiple receivers in the group, leading to the danger that a malicious packet can potentially affect multiple receivers. To prevent such malicious behaviors, the receiver should be able to verify that the received data really originates from the claimed source and was not modified during transmission (i.e., source authentication).

Due to the resource limitation, it is challenging to ensure source authentication for the in-vehicle network and the real-time performance requirement. Overall, except for the source authentication property, the proposed design should satisfy the following two design goals:

1. *Lightweight sender*: the proposed design should not incur high overhead at the sender side during the transmission of broadcast packets. This is important for in-vehicle communication, because ECUs often have very limited computational resources (e.g., 160 MHz CPU), as mentioned in § 4.4.1. The high overhead at the sender side will undoubtedly produce low-throughput traffic with high latency.

2. *Overall efficiency*: although the proposed design will introduce overhead, the overall performance should be optimized accordingly, because of the importance of the timely delivery of safety-critical packets.

To meet these two design goals, GATEKEEPER is built upon two insights. First, symmetric cryptography is much faster than asymmetric cryptography, which is common knowledge and is also demonstrated in § 4.6.3. Therefore, we mainly utilize symmetric cryptography in GATEKEEPER to build a lightweight sender. Second, the gateway/the domain controller in an in-vehicle network usually has more computational resources than in-vehicle ECUs. For example, NXP MPC-LS-VNP-RDB [196] gateway includes four Arm Cortex-A53 64-bit processors, with up to 1.4 GHz CPU speed, while NXP MPC5748G [195]) ECU board only has two 160 MHz Power e200Z4 32-bit CPUs and one 80 MHz Power e200Z2 32-bit CPU. To optimize overall efficiency, we thus let the gateway or the domain controller handle resource-consuming operations such as cryptography functions.

Note that, using symmetric cryptography, like HMAC, at the sender side cannot ensure source authentication in broadcast communication. As we discussed in §4.4.2, existing protocols (e.g., MACsec, IPsec, and TLS) do not provide source authentication [63] for broadcast communication. Since all participants in a broadcast group have the same symmetric key for communication, any malicious participant in a communication group can impersonate the sender and forge packets to other receivers. However, in unicast communication, source authentication can be achieved through the symmetric message authentication code (MAC). While sending a packet, the sender attaches a MAC with the packet. In this case, the receiver can verify the correctness of the MAC to verify the packet source identity.

Based on the discussion above, we introduce an on-path authenticator (Figure 4.2), which co-locates with the switch/router. The broadcast can be either inter-domain or intra-domain; therefore, the authenticator can be the gateway or the domain controller, respectively. In a switched Ethernet network, a broadcast packet is first sent to a switch/router via a unicast channel. Then, the switch/router forward duplicated packets to broadcast receivers. For a unicast channel, it is

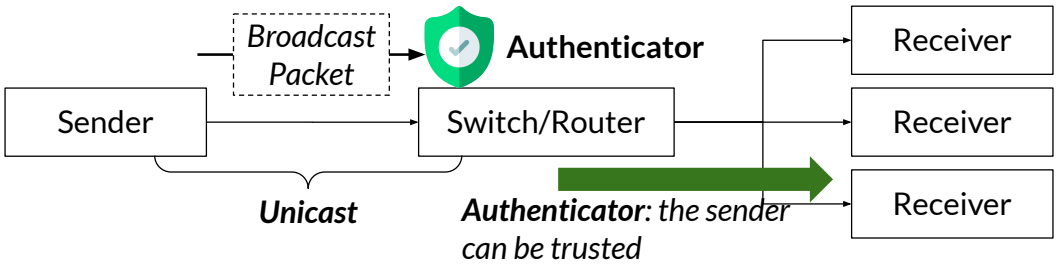


Figure 4.2: The overview design of GATEKEEPER.

possible to ensure source authentication. Therefore, while observing a broadcast packet, the authenticator can first verify the sender's identity. Then, the authenticator can construct another MAC for the broadcast packet and embed it in a *proof packet* for each receiver. In this way, we delegate the sender's identity verification to the authenticator. Once the receiver accepts the proof packet, the receiver can verify that the broadcast packet is successfully forwarded by the authenticator and thus carries the correct sender's identity. Table 2.1 further highlights the low time and bandwidth overhead of the design of GATEKEEPER, compared to existing works [107, 133, 64].

Key Materials. To protect the broadcast communication, we assume that a group key, $\mathcal{K}_G$, is shared among all senders and receivers based on a key management mechanism [69], which is outside the scope. For each broadcast participant, while joining the communication group, it initiates a secure unicast channel and negotiates a shared secret key $\mathcal{K}_U$ with the authenticator, following the same handshake protocol of existing security protocols (e.g., MACsec, IPsec, TLS). For the sake of simplicity, we differentiate the notation of the sender's key $\mathcal{K}_{US}$ and the receiver's key $\mathcal{K}_{UR}$ in this chapter.

4.5.2 GATEKEEPER Roles

There are three different roles in GATEKEEPER: (1) sender (S), (2) receiver (R), and (3) authenticator (A). We will describe detailed steps along with three roles in the following paragraphs.

Sender. At the sender side, the sender initializes a packet counter cnt and sends it with the packet. For each outgoing broadcast packet, the sender attaches a traffic type and a message

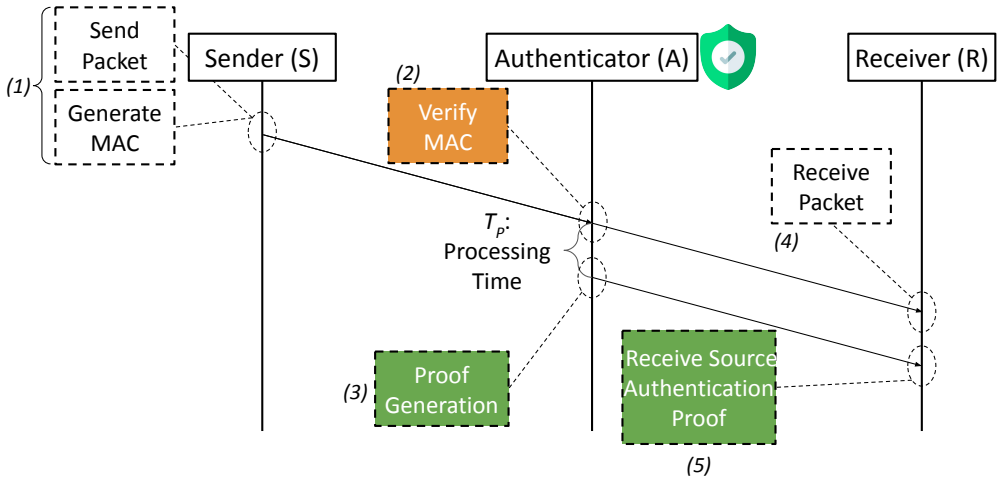


Figure 4.3: Detailed workflow of GATEKEEPER.

authentication code (MAC):

$$pkt' = f(\mathcal{K}_G, cnt \parallel pkt)$$

$$mac_S = MAC(\mathcal{K}_{US}, type \parallel pkt')$$

$$\text{S} \rightarrow \text{A} : type \parallel pkt' \parallel mac_S$$

where f can either encrypt the outgoing packet or attach a MAC for integrity protection, depending on the application scenarios. For the next outgoing packet, the local counter cnt will be incremented.

Authenticator. In GATEKEEPER, the authenticator aims at verifying the identity of the sender and informing receivers. The functionality of the authenticator is also described in Algorithm 1.

When the authenticator observes a broadcast packet, it picks the corresponding key $\mathcal{K}_{US}$ to verify the incoming packet. If the authenticator cannot find a key that corresponds to the sender-authenticator unicast channel, the sender may not be involved in any broadcast communication. In this case, no proof packet will be generated. The authenticator can also drop the incoming packet to prevent insecure communication.

If the authenticator finds the secret key $\mathcal{K}_{US}$, it will verify the correctness of mac_S in the packet.

Algorithm 1: GATEKEEPER authenticator.

Input: Packet $type \parallel pkt' \parallel mac_S$, Sender S

$key = findKey(S)$;
if $key == NULL$ *or* $mac_S != MAC(key, type \parallel pkt')$ **then**
    drop the incoming packet;
    return;
end
forward $type \parallel pkt'$ to intended receivers;
$h' = HASH(type \parallel pkt')$;
foreach *receiver* R *and* $\mathcal{K}_{UR}$ **do**
    $mac_A = MAC(\mathcal{K}_{UR}, h')$;
    $proof = h' \parallel mac_A$;
    send $proof$ to R;
end

If the mac_S is incorrect, no proof packet will be generated, and the packet will be dropped. Note that, the authenticator can remove mac_S while forwarding it to the receiver, as mac_S is not needed for the receiver.

For each receiver R, the authenticator uses the corresponding $\mathcal{K}_{UR}$ to calculate a MAC, mac_A, over the hash h' of the original secured packet pkt'. After that, the authenticator constructs a proof packet $h' \parallel mac_A$ and sends it to R.

Receiver. While receiving the broadcast packet pkt' with the traffic type $type$ from the sender, the receiver first checks the correctness of it using $\mathcal{K}_G$:

$$cnt \parallel pkt = f^{-1}(\mathcal{K}_G, pkt')$$

where f^{-1} is the corresponding decryption or verification function of f. If it is correct, the receiver should calculate the hash of pkt' and store the packet pkt locally. Then, the receiver needs to inspect the counter cnt, by comparing it with the local counter for the corresponding sender. If cnt is smaller than the expected counter value, the receiver should discard the packet, because a replay or spoofing attack likely happens. If two values match, the receiver will increase the local counter for the sender by 1. Note that, cnt may be greater than the expected counter value, due to the packet loss. In this case, the receiver will still accept the packet if cnt is higher than the expected

counter value within a threshold. We follow existing works to set the threshold to 1 [171].

Once the receiver receives the proof packet, the receiver looks for a corresponding packet based on the hash value h'. If no matching packet is found, the proof packet should be discarded. Then, the receiver uses $\mathcal{K}_{UR}$ to verify the proof packet. If the proof packet is incorrect (i.e., $MAC(\mathcal{K}_{UR}, h')$ does not equal to the received mac_A), the packet and the proof packet should both be discarded. To this end, we successfully ensure the source authentication, and the receiver can now process the broadcast packet pkt.

Similarly, the loss of proof packets could happen, resulting in a large number of stored but not processed packets at the receiver side. Thus, the receiver will periodically purge the stored "expired" packets, which are defined as packets that have already violated the maximal latency requirement of the corresponding traffic type. The receiver can periodically compare the current timestamp and the receiving timestamp of each stored packet. If the difference between two timestamps is greater than the allowed maximal latency, the receiver can discard and delete the stored packet.

4.5.3 Denial-of-Service Protection

Since we assume that the attacker can compromise an in-vehicle device (e.g., ECU), the attacker can intentionally send a large number of broadcast packets to the authenticator with the goal of triggering the proof generation process. To defend against the DoS threat, we propose a countermeasure based on the time-lock puzzle [185] (Figure 4.4). The proposed approach has three benefits [208]: (1) lightweight puzzle generation and verification, (2) linear granularity delay time control, and (3) non-parallelizable solving process, while existing countermeasures have several limitations. For example, both one-way hash chain [135] and hash-based reversal puzzle [56, 101] cannot achieve fine-grind control over delay time. Besides, both of them require exhaustive searching of a pre-image to solve the puzzle, which is a parallelizable task. Moreover, the puzzle verification of the two types of puzzles above requires one hash operation, while our approach does not.

When a DoS attack behavior is detected, the authenticator will generate and send out a time-lock puzzle. Meanwhile, the authenticator will not forward any traffic from this malicious sender until the sender sends back the correct solution. The puzzle generation and verification are lightweight for the authenticator, while there is no shortcut for the sender to solve a time-lock puzzle. By controlling the time to solve the puzzle, the authenticator can slow down the malicious sender.

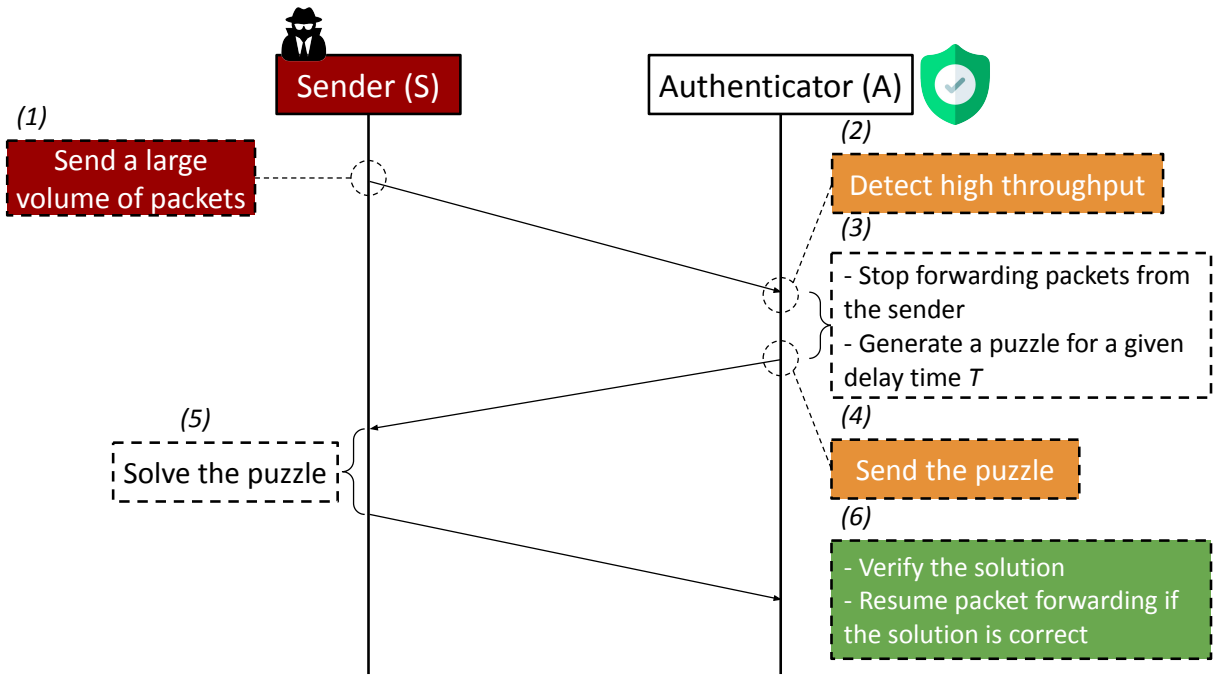


Figure 4.4: DoS protection workflow of GATEKEEPER

DoS signal. First of all, the authenticator needs to identify the DoS behavior (i.e., sending a large volume of packets). To achieve this goal, we attach a *packet type* field to the outgoing packets of each sender. By comparing the performance profile, like Table 4.1, the authenticator is aware of the expected throughput for a given packet type. Since the authenticator co-locates with the switch/router, the real-time statistics about the packet forwarding can be calculated easily, by measuring the throughput every time interval l. Suppose the monitored throughput w_m is greater than the expected throughput w_e. In that case, the authenticator will consider the corresponding sender as a potential DoS attacker, leading to the generation of the time-lock puzzle. We admit that this is a rough signal for DoS detection and may result in false positives. However, the DoS detection is

orthogonal to the DoS protection approach and can be replaced by more precise versions.

Puzzle generation. While generating the time-lock puzzle, the authenticator (i.e., the puzzle generator) seals a randomly generated message $\mathcal{M}$ in the puzzle and enforces the penalty delay time T. The penalty delay time T must follow the inequality $\frac{w_m(l-T)}{l} \leq w_e$, where the measured throughput in the next time interval is no greater than the expected throughput w_e. That is, T should be at least $(1 - \frac{w_e}{w_m})l$. In addition, the authenticator needs to know the number of squarings per second that can be performed by the malicious sender, notated by S, which can be learned and configured in the manufacturing time.

Algorithm 2: Time-lock puzzle generation.

Input: Penalty delay time T, The number of squarings per second of the sender S, Solution message $\mathcal{M}$

Output: Time-lock puzzle $(n, a, t, C_{\mathcal{K}}, C_{\mathcal{M}})$

1 randomly generate two secret primes p and q;
2 $n = pq$;
3 $t = TS$;
4 randomly generate a key $\mathcal{K}$;
5 $C_{\mathcal{M}} = Enc(\mathcal{K}, \mathcal{M})$; // encrypt the solution
// using the shortcut to calculate $C_{\mathcal{K}}$
6 randomly generate a, where $1 < a < n$;
7 $\phi(n) = (p-1)(q-1)$;
8 $e = 2^t \pmod{\phi(n)}$;
9 $b = a^e \pmod{n}$;
10 $C_{\mathcal{K}} = \mathcal{K} + b \pmod{n}$; // calculate $C_{\mathcal{K}} = \mathcal{K} + a^{2^t} \pmod{n}$

As shown in Algorithm 2, the authenticator first calculates a composite modulus n as the product of two large randomly-generated primes p and q. Then, the authenticator derives the total number of squarings t from the penalty delay time T and sender's solving speed S. After that, the pre-determined solution message $\mathcal{M}$ will be encrypted (e.g., AES), using a random key $\mathcal{K}$. At last, the authenticator will hide the key $\mathcal{K}$, by calculating $C_{\mathcal{K}} = \mathcal{K} + a^{2^t} \pmod{n}$, where a is a random number between 1 and n (exclusively). Since the authenticator knows p and q, the Euler's totient function $\phi(n)$ can be computed, which enables the efficient computation of $C_{\mathcal{K}}$ (Line 7-10 in Algorithm 2) [185].

Puzzle solving. After receiving the generated time-lock puzzle, the malicious sender (i.e., the

puzzle solver) needs to recover the key $C_{\mathcal{K}}$ and then decrypt the sealed solution message $\mathcal{M}$. For a secure encryption algorithm, it is non-trivial to brute force the key; thereby, one feasible approach is to calculate $b = a^{2^t} \pmod{n}$ and derives $\mathcal{K}$. However, without knowing the factorization of n (i.e., p and q), it is provably hard to compute $\phi(n)$ from n [185]. That is, there is no efficient way to compute b, and the only solution is to perform t squaring operations sequentially (Line 1-5 in Algorithm 3).

Algorithm 3: Time-lock puzzle solving.

Input: Time-lock puzzle $(n, a, t, C_{\mathcal{K}}, C_{\mathcal{M}})$
Output: Solution $\mathcal{M}$

```
// repeated squarings to calculate b = a^(2^t) (mod n)
```
1 $b = a \pmod{n}$;
2 **while** $t \neq 0$ **do**
3 $\quad b = b^2 \pmod{n}$;
4 $\quad t = t - 1$;
5 **end**
```
// recover the key
```
6 $\mathcal{K} = C_{\mathcal{K}} - b \pmod{n}$;
7 $\mathcal{M} = Dec(K, C_{\mathcal{M}})$; // decrypt the solution

Puzzle verification. Puzzle verification is relatively easy, and the authenticator only needs to compare the stored solution with the received solution. If two values are equal, the authenticator will resume forwarding the traffic from the sender. Otherwise, the authenticator will keep blocking the malicious sender.

4.5.4 Formal Verification

To show how GATEKEEPER prevents attacks listed in §4.3, we have formally modeled GATEKEEPER and verified security properties using the Tamarin prover [142]. The Tamarin prover has been widely used for security protocol verification (e.g., TLS 1.3 [49, 48], 5G [21, 47], WPA2 [50], IoT protocols [106], and SecOC [118]), which supports unbounded verification of symbolic models. Using the Tamarin prover, we verify three security properties for GATEKEEPER: (1) no packet replay, (2) source authentication, and (3) DoS protection. Note that, we only model the design of

GATEKEEPER as mentioned in § 4.5 and do not include any existing security protocols.

```
lemma NoReplay:
  " All receiver pkt #i #j.
     Process(receiver, pkt) @ #i
    & Process(receiver, pkt) @ #j
     ==> #i = #j"
```

Listing 4.1: No replay attack property

For the first property, Listing 4.1 specifies that two identical packet processing events cannot happen at the same time (`#i` and `#j` denotes two timepoint variables in Tamarin). Due to the use of the counter, we can ensure the receiver will not process the same packet twice, as the counter will change after packet processing.

```
lemma SourceAuthentication:
  " All receiver authenticator pkt r_key #k #n.
     Process(receiver, pkt) @ #k
    & KeyAccepted(receiver, authenticator, keyr) @ #n
    ==> (
          (Ex sender1 sender2 keys1 keys2 #i #j #l #m.
            Send(sender1, pkt) @ #i
           & Send(sender2, pkt) @ #j
           & KeyAccepted(sender1, authenticator, keys1) @ #l
           & KeyAccepted(sender2, authenticator, keys2) @ #m
           & (sender1 = sender2) & (#i = #j) & (#l = #m)
           & (#i < #k)
          )
        | (Ex sender #x. RevealLtk(sender) @ #x )
        | (Ex #y. RevealLtk(receiver) @ #y )
        | (Ex #z. RevealLtk(authenticator) @ #z )
       )"
```

Listing 4.2: Source authentication property

For the second property, with the help of the Tamarin prover, we show that the attacker is unable to obtain the group key $\mathcal{K}_G$ and the shared key $\mathcal{K}_U$ between the authenticator and the sender/receiver, unless the attacker compromise an ECU (§ 4.3). Even with both keys, the attacker cannot impersonate others, because $\mathcal{K}_U$ is only bound with a single identity during the negotiation. In this case, we can verify that, for each processed packet at the receiver side, only one legitimate sender produced the corresponding packet before (Listing 4.2). That is, the source authentication

property of GATEKEEPER is validated.

```
lemma DoSProtection:
  " All authenticator #i.
     ProofGeneration(authenticator) @ #i
    ==> (
          (Ex sender #j.
            Send(sender, pkt) @ #j
           & Honest(sender) @ #j
           & #j < #i)
         | (Ex sender #r.
             RevealLtk(sender) @ #r
           & Honest(sender) @ #i
           & #r < #i)
       )"
```

Listing 4.3: DoS protection property

For the last security property, we actually prove a *weakened* property that only the *honest* sender can trigger the generation of proof packets, unless the sender is compromised by the attacker (i.e., `RevealLtk(sender)` in Listing 4.3). An honest sender is defined as a sender whose traffic throughput is no greater than the claimed throughput in the packet header. While solving the time-lock puzzle, the statistical throughput value will undoubtedly drop. We therefore consider the sender as an honest sender, if the sender sends back the correct solution. Then, we follow an existing work [106] to mark the proof generation as a cryptographically heavy operation and verify that only the sending event from an honest sender can trigger such heavy operations. It is worth mentioning that, for the last property, we only verify if the DoS protection mechanism can be triggered rather than ensure it can quantitatively slow down any malicious senders. This limitation is inherited from the verification tool, Tamarin prover, as it can only produce a binary answer (yes or no) for each specified property. Other verification tools, like the probabilistic model checker [111], can further help generate quantitative verification results.

4.6 Evaluation

In this section, we first present the performance evaluation results of MACsec, IPsec, and TLS in a resource-restricted environment. Unfortunately, none of them can fully satisfy the performance requirements (§4.4.1). Besides, we implement our prototype of GATEKEEPER on a Docker-based testbed. Our simulation results confirm that GATEKEEPER incurs low latency overhead, can support the transmission of high-throughput LiDAR traffic, and significantly outperforms TESLA [169, 168, 170, 167].

We want to answer the following research questions:

- **RQ0**: Can Docker-based testbed produce a similar performance to ECUs?
- **RQ1**: Can existing security protocols satisfy the performance requirement of the in-vehicle communication?
- **RQ2**: What is the performance overhead introduced by Gatekeeper?
- **RQ3**: Can Gatekeeper scale well if there are multiple broadcast receivers?
- **RQ4**: What is the performance overhead of the time-lock puzzle generation and solving?

4.6.1 Testbed Setup

Our experiments are conducted on a Docker-based testbed, calibrated to simulate a resource-restricted environment of the ECU. The Docker-based testbed offers two significant benefits. First, Docker allows us to control the resource constraints (e.g., memory, CPU) of a Docker container to simulate a resource-limited environment. Second, we can reuse off-the-shelf protocol implementations in the container. The development boards of the in-vehicle devices are often microcontroller boards (e.g., NXP DEVKIT-MPC5748G), running a real-time operating system (e.g., FreeRTOS) rather than Linux. We only found one available TLS implementation (i.e., TLS 1.3 in WolfSSL), while MACsec and IPsec are missing. Besides, it is laborious and error-prone to implement IPsec

and MACsec on our own. Notably, our goal is to evaluate security protocol candidates and GATEKEEPER prototype on a comparable platform to the in-vehicle network. Thereby, we stick to the Docker-based testbed.

Docker-based Testbed. All Docker containers run on a host machine (Ubuntu 18.04, Linux kernel 5.4.0-77) with six cores clocked at 3.8 GHz and 16 GB memory. The Docker image uses Ubuntu 18.04 and installs all necessary applications. For TLS 1.3, we choose WolfSSL (v4.4.0), a TLS implementation for embedded systems, also used by our development board, DEVKIT-MPC5748G. IPsec and MACsec communication are provided by the underlying Linux kernel of the host machine. For IPsec handshake, we utilize strongSwan (v5.6.2). For MACsec handshake, we install hostapd and wpa_supplicant (v2.9 for both). Furthermore, for performance measurement, `iperf3` (v3.9) is added to the Docker image as well. We also integrate WolfSSL into iperf3 so that we can measure the performance of TLS communication.

We instantiate Docker containers for the sender, gateway, and multiple receivers, and only restrict the resources of the sender and receiver containers to simulate ECUs. Both sender and receiver containers are connected with the gateway container through virtual Ethernet interfaces (100 Mbps links). In the gateway container, we create and configure a bridge device to forward packets between sender and receiver containers.

Development Boards. The development board we use, NXP DEVKIT-MPC5748G [195], has two 160 MHz e200Z4 CPUs and one 80 MHz e200Z2 CPU. The memory resource is also limited, with only 6 MB Flash and 768 KB on-chip SRAM. In the following experiments, the development board is mainly used to calibrate the resource restrictions of the Docker container. We measure the performance of the cryptography algorithms on the board and calibrate the parameters of the container to produce similar results.

4.6.2 Prototype of GATEKEEPER

We implement three different roles of GATEKEEPER (§4.5.2) in C, as shown in Figure 4.5. As a proof of concept, we currently place all three roles at Layer 2. Note that, GATEKEEPER can also

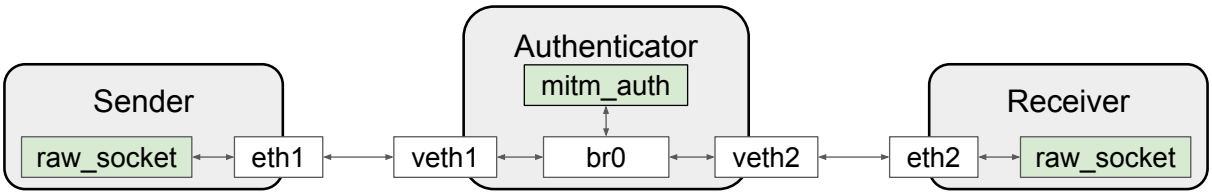


Figure 4.5: GATEKEEPER prototype.

be implemented at any layer that supports multicast/broadcast communication.

We choose HMAC-SHA256 and SHA256 to realize the MAC and $HASH$ functions that are presented in §4.5.2. At the sender side, we utilize Layer 2 raw socket to append the extra message authentication code to the outgoing broadcast packet (§4.5.2). Also, the Layer 2 raw socket is used in the receiver container to monitor the broadcast packet and the proof packet. In the authenticator container, a kernel module, named `mitm_auth`, is loaded to monitor all traffic forwarded by the bridge device. The kernel module will wrap the generated proof packet in an Ethernet frame with a self-defined Ethernet type.

4.6.3 RQ0: Performance Calibration

This subsection is a prerequisite for other experiments. We first evaluate cryptography algorithms on the development board, as summarized in Figure 4.6. Then, we pick AES-128-GCM as the reference to calibrate the resource restrictions of the container because AES-GCM is supported by all three security protocols.

The calibrated results are presented in Figure 4.7, showing that the answer to RQ0 is positive. For symmetric ciphers and hash functions, we measure their throughput of encryption, hash generation, or message authentication code generation. To achieve similar results to the development board, the sender and receiver containers are configured to only use 1% CPU resources of one CPU core. We set the memory resource to 50 MB, as the container itself requires more memory than the development board.

According to the results in Figure 4.7a, we note that such a resource-limited environment can hardly satisfy the throughput requirements listed in Table 4.1. Even the best performing ChaCha20-

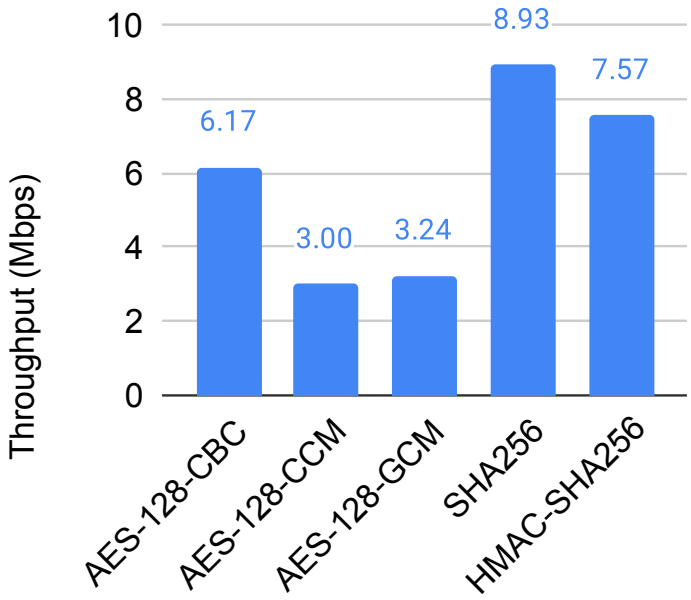
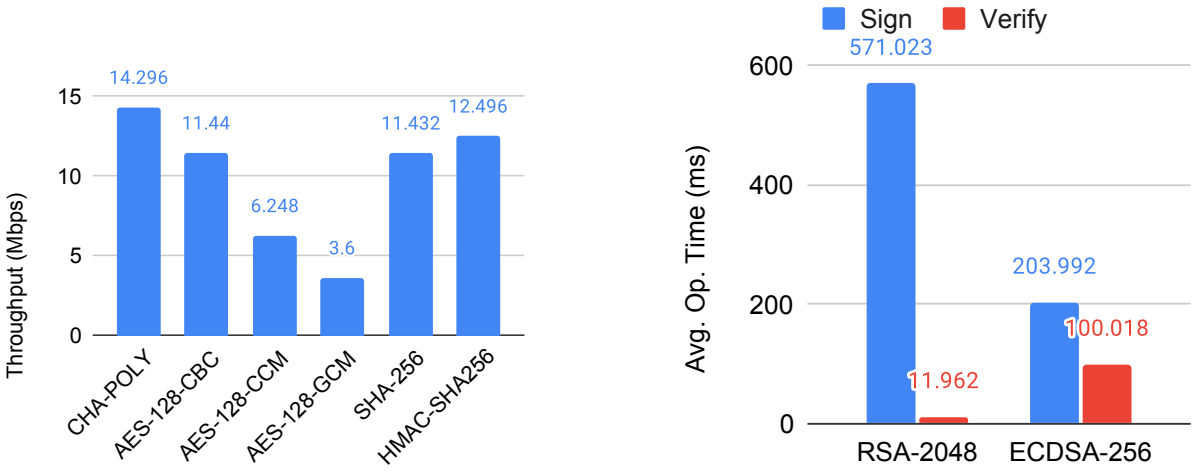


Figure 4.6: Performance of symmetric cipher suites and hash functions on the development board.



(a) Symmetric ciphers and hash functions

(b) Asymmetric algorithms

Figure 4.7: Performance of cryptography algorithms in the Docker container.

Poly1305 cipher suite can only achieve 14.296 Mbps processing throughput. Apparently, the development board needs more resources to support the transmission of Camera and LiDAR data. Figure 4.7b illustrates the operation time of signature generation and verification with two different asymmetric algorithms: RSA and ECDSA. Overall, both algorithms incur high overhead. The ECDSA algorithm performs better in the signing procedure with timing of 203.99 ms, but the verification is around ten times slower than RSA. The slow performance of asymmetric cryptography indicates the impossibility of adopting them at the sender side to enforce source authentication.

4.6.4 RQ1: Performance of Security Protocols

We measure the handshake and communication performance of three security protocols. For handshakes, we configure three protocols to use elliptic-curve cryptography (ECC) certificate with a 256-bit ECC key. Each protocol will be initiated 100 times. For communication performance, we run iperf3 along with three protocols to evaluate their throughput. All three protocols use the same symmetric cipher, AES-128-GCM, to encrypt the transmitted data, as this is the only cipher that is supported by three protocols.

Table 4.3 presents the handshake time, the total size of transmitted packets during the handshake, and the communication throughput. TLS 1.3 performs the best among three candidates, and IPsec is comparable to TLS 1.3. The overall handshake performance of MACsec is worse than other protocols, because MACsec spends lots of time on MACsec Key Agreement (MKA), which is the second stage of its handshake procedure. As expected, their communication throughput cannot satisfy the throughput requirements for Camera, LiDAR, and Video data, as the throughput is bounded by the chosen cipher suite (i.e., AES-128-GCM in Figure 4.7a). For performance improvements, we will discuss the benefits of hardware cryptography acceleration and lightweight cryptography algorithms in § 4.7.

Protocol	Handshake		Throughput (Mbps)
	Time (s)	Packet Size (bytes)	
MACsec	3.27	5048	2.87
IPsec	1.11	2769.1	2.94
TLS 1.3	0.80	2036.5	3.42

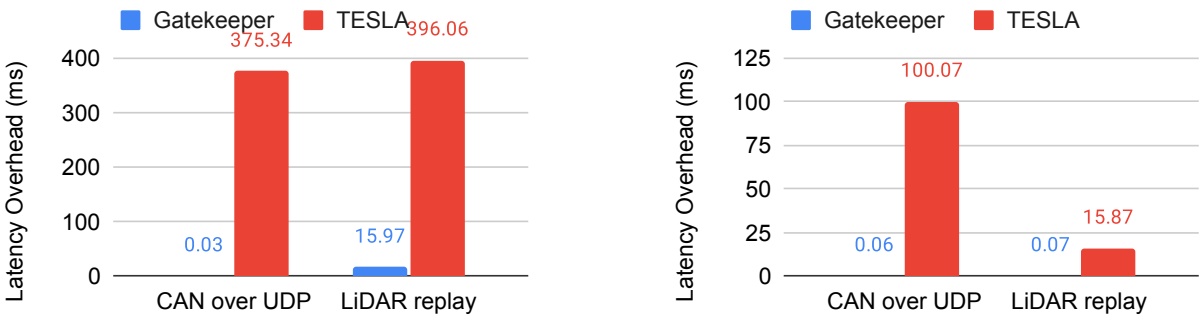
Table 4.3: Handshake and communication performance.

4.6.5 RQ2: Latency Overhead of GATEKEEPER

To demonstrate the effectiveness of GATEKEEPER, we evaluate GATEKEEPER under two scenarios: (1) CAN over UDP; (2) LiDAR replay over UDP. We think both scenarios are representative: (1) the first one represents the most commonly used CAN broadcast scenario for the in-vehicle network; (2) the second one represents the transmission of newly emerging LiDAR sensor data for the autonomous vehicle. For the first scenario, we replay randomly generated CAN frames every 100 ms. For the second scenario, we replay pre-recorded LiDAR data [216] over UDP, following its original timing. As discussed in § 4.6.3 and § 4.6.3, without hardware acceleration or additional computational resources, no security protocol can satisfy the throughput requirement of the large-volume traffic, like LiDAR. Therefore, we focus on the *latency requirement* in this subsection. We measure the end-to-end latency of each transmitted packet and calculate the latency overhead to see if GATEKEEPER can satisfy the *latency* requirements in Table 4.1. Note that, we only instantiate one receiver container in this set of experiments. The scalability evaluation over multiple receivers is shown in § 4.6.6.

Besides, we compare GATEKEEPER with TESLA in the two evaluation scenarios. For a fair comparison, we replace MD5 and HMAC-MD5 in the original TESLA prototype with SHA256 and HMAC-SHA256, which are used in GATEKEEPER. The interval duration of TESLA is set to 10 ms. We then follow TESLA's paper [168] to set up the key disclosure delay as 2. In this way, we can uncover the performance differences caused by different design choices.

Figure 4.8a shows that GATEKEEPER significantly outperforms TESLA. The latency overhead of the LiDAR replay scenario is even larger, which is due to the large throughput (18 Mbps) and



(a) Under resource-limited environments

(b) Given enough computational resources

Figure 4.8: Latency overhead of GATEKEEPER and TESLA.

high packet rate (2000 Hz) of LiDAR traffic (Table 4.1). On the other hand, the low performance of SHA256 and HMAC-SHA256 also increase the latency. Nonetheless, the absolute latencies of GATEKEEPER under two evaluation scenarios are 0.1 ms and 15.98 ms, respectively, which are within the latency requirements of Data 1 and LiDAR.

One major source for the high latency overhead of TESLA is the time-delayed key disclosure embedded in the design of TESLA. The receiver cannot verify and process the broadcast packet until it receives the key disclosure. However, the key disclosure is only available after at most 20 ms, based on our current settings, and is always attached to the next outgoing packet. If the packet rate is greater than 20 ms, the latency incurred by TESLA can become even higher. GATEKEEPER does not rely on delayed key disclosure and thus can avoid such high latency.

4.6.6 RQ3: Scalability of GATEKEEPER

We evaluate the scalability of GATEKEEPER to understand if GATEKEEPER can still produce low latency when there are multiple receivers. Like the previous section, we measure the transmission latency of CAN traffic, while increasing the number of receivers to 2, 4, 8, and 16. We do not include TESLA in this subsection, because its latency overhead, by design, will not change with the number of receivers. Readers can refer to Figure 4.8a for comparison. Figure 4.9 shows the average latency for all sender-receiver pairs with and without GATEKEEPER.

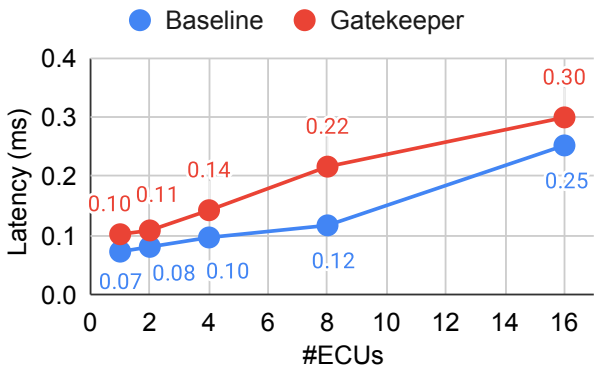


Figure 4.9: Transmission latency of CAN traffic: linear scaling.

As shown, GATEKEEPER scales well, because the latency caused by GATEKEEPER increases *linearly* with the number of receivers. GATEKEEPER starts to violate the latency requirement of CAN traffic (i.e., 0.1 ms for Data 1 in Table 4.1), when there are 4 receivers. However, it is worth noting that even the baseline (i.e., w/o GATEKEEPER) cannot satisfy this latency requirement with 8 receivers. The inefficiency of the baseline scenario is due to the performance limitation of the software-based bridge interface in our testbed. In the real-world deployment of an in-vehicle network, we would have a hardware-based switch/router (e.g., NXP MPC-LS-VNP-RDB [196]), which significantly accelerates the performance of packet forwarding. Thereby, we should focus on the latency overhead incurred by GATEKEEPER, which is still lower than TESLA's latency overhead with increasing number of receivers.

4.6.7 RQ4: Performance of Time-lock Puzzle

In this subsection, we evaluate the puzzle generation time at the authenticator container and the puzzle solving time at the sender container. We first measure the number of squarings per second S in the sender container, which is 260 in our testbed, and use AES-128-CBC for the encryption algorithm. As shown in Figure 4.10, the puzzle generation is efficient, which only takes the authenticator around 1.6 ms. On the other hand, solving a time-lock puzzle costs the sender a relatively long time, which has a linear correlation with the input parameter T of the puzzle generation (Algorithm 2). Note that, the input parameter T of the puzzle generation is a continuous value, unlike

the discrete hardness level for the hash-based puzzle. Thus, the authenticator can precisely control the penalty delay time for the attacker.

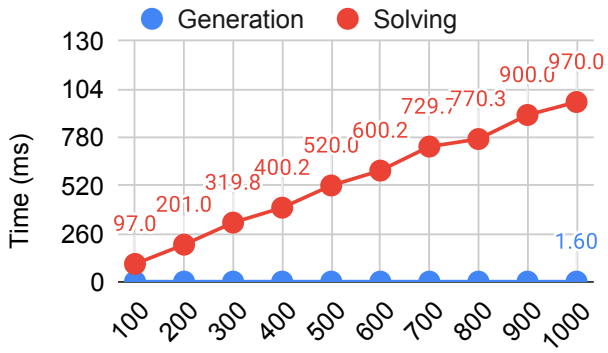


Figure 4.10: Performance of the time-lock puzzle generation and solving. The x-axis is the input parameter T for puzzle generation.

4.7 Discussion

Attacker's capability. Beyond our threat model, a stronger attacker can compromise the gateway. We can hardly ignore this worst case of the gateway compromise, because the gateway can become the single point of failure in the assumed network architecture 4.2.1. What's worse for GATEKEEPER, a malicious gateway can collude with a compromised ECU to still launch the spoofing attack. We also extend the formal model of GATEKEEPER to cover the gateway compromise and confirm this consequence (i.e., violating source authentication). This worst-case scenario should engage an OEM's interests to eliminate gateway attackers. We highly suggest the manufacturer enforce advanced protection mechanisms, such as Hardware Security Module (HSM) and Trusted Execution Environment (TEE), for the gateway and safety-critical ECUs. These hardware security features can bring additional protection for sensitive key materials or safety-critical applications (e.g., GATEKEEPER) [76].

Hardware acceleration and lightweight cryptography. As indicated in § 4.6.3 and § 4.6.4, the runtime throughput of security protocols is bounded by the chosen cryptography algorithm. Our benchmark results indicate that resource-limited ECUs produce low performance of cryptog-

raphy algorithms (Figure 4.6 and 4.7). For performance improvement, one option is to enable hardware acceleration. The NXP MPC5748G board has a hardware security module (HSM) installed. With HSM enabled, the throughput of AES-128-GCM can be accelerated from 3.24 Mbps to 81.42 Mbps. Since we do not have similar hardware acceleration on the host machine, we only use software-based cryptography implementations for evaluations. Another option is to adopt lightweight cryptography (LWC) algorithms that are suitable for use in constrained environments. NIST [156] has initiated a process to solicit lightweight cryptographic algorithms for years, because the performance of current NIST cryptographic standards is not acceptable for constrained devices. Their benchmark results on 32 candidates [155] further highlight the potential of LWC for performance improvements. For example, on the ARM Cortex-M0+ platform, about half of the candidates show performance improvement over AES-GCM. Therefore, for real-world deployment, the manufacturer should consider enabling the hardware acceleration or adopting lightweight cryptography algorithms if needed.

Asymmetric cryptography. The design goals of GATEKEEPER include (1) lightweight sender and (2) overall efficiency. Thus, we relocate computationally heavy operations to the authenticator. Inspired by the imbalanced performance of digital signature signing and verification, can we replace the HMAC-based proof generation with the digital signature? In this case, the authenticator only needs to generate one digital signature and broadcast it to all receivers. Unfortunately, the answer to this question is still negative. We replace HMAC generation and verification at the authenticator and receiver side with RSA 2048 signing and verification operations, respectively. As demonstrated in Figure 4.11, the end-to-end latency of GATEKEEPER w/ RSA 2048 is around 33 ms, which does not suit the transmission of safety-critical data. Nonetheless, we admit that the constant scalability, in both TESLA (§ 4.6.6) and asymmetric cryptography, is conceptually better than GATEKEEPER. Users should adopt TESLA or asymmetric cryptography to avoid unnecessary operations (e.g., calculating HMAC for each duplicated broadcast packet), if the latency overhead of them can be reduced further.

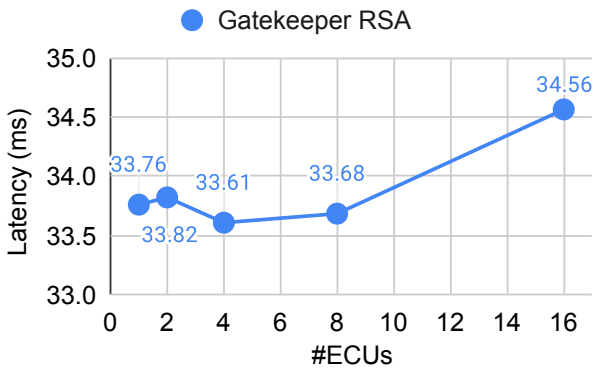


Figure 4.11: Latency of Gatekeeper w/ RSA 2048 (baseline is the same as Figure 4.9)

4.8 Conclusion

In this chapter, we revisit three existing security protocols, MACsec, IPsec, and TLS, in the context of in-vehicle Ethernet. This chapter serves as an initial guidance for the in-vehicle Ethernet, users can choose the security protocol on their own based on their uses cases and evaluations. Overall, TLS and IPsec are suitable for inter-domain communication, while MACsec is favorable for intra-domain communication. Except for the applicablity of securing intra-domain communication, another reason of including MACsec is that AVB/TSN [88, 89, 85, 84] protocols directly work above Layer 2 and cannot be protected by TLS or IPsec. In summary, we believe that a combination of TLS and MACsec are preferred candidates for in-vehicle Ethernet, but there are use-cases for IPsec as well. Besides, users can adopt GATEKEEPER for broadcast authentication, if DoS prevention is desired, and implement GATEKEEPER on the domain controller and the gateway. Our evaluation results demonstrate that GATEKEEPER incurs low latency overhead and significantly outperforms TESLA for both CAN and LiDAR traffic. However, if users have concerns about using GATEKEEPER or if there are only a few receivers, users should consider using several concurrent unicast to ensure source authentication for broadcast communication.

To adopt the existing security protocol for the automotive in-vehicle Ethernet network, we summarize the deployment recommendations for different use scenarios, as specified below and also in Table 4.4. Note that, for each security protocol candidate, we enumerate implementation re-

quirements in the following bullet lists, in which items correspond to four security requirements: integrity, optional confidentiality, authenticity, and anti-replay. Other security requirements (e.g., key materials, end-to-end protection, source authentication, and DoS prevention) are either implemented by design or unsupported features of security protocols, so we do not discuss them in the following parts. Readers can refer to § 4.4.1 for more details about the justification of these security requirements.

Scenarios		Integrity	Optional Confidentiality	Authenticity	Anti-replay	Others
Multiple LANs	TLS	AEAD, Appendix B.4 in RFC 8446 [183], Section 4.1.2.4 of RFC 6347 [184]		Handshake protocol, Section 4 in RFC 8446 [183], Section 4.2 of RFC 6347 [184]	Per-record nonce, Section 5.3 in RFC 8446 [183]	- RFC 7525 [198] - RFC 8996 [147]
	IPsec	AH, RFC 4302 [103]	ESP, RFC 4303 [104]	IKEv2, RFC 7296 [102]	Sequence number, Section 3.4.3 in RFC 4302 [103]	- No tunnel mode
Single LAN	MACsec	Default cipher suite, Clause 14 in IEEE 802.1AE-2018 [86]		MKA, IEEE 802.1X-2020 [87]	Packet number, Clause 10.6 in IEEE 802.1AE-2018 [86]	- Hardware-based implementation - Shorter MKA interval, 0.5 s

Table 4.4: Deployment recommendations under different scenarios (AEAD: Authenticated Encryption with Associated Data; AH: Authentication Header; ESP: Encapsulating Security Payload; MKA: MACsec Key Agreement)

Multiple LANs with TLS. First, we consider the assumed network architecture in § 4.2.1. Since the domain-based in-vehicle network involves both Layer 2 (L2) and Layer 3 (L3) protocols, users should choose a security protocol that works at Layer 3 or above to ensure the end-to-end security. TLS serves as the top candidate, because the performance evaluation (§ 4.6.4 and Table 4.3) indicate that TLS outperforms the other two protocol candidates on both the handshake and communication performance. To satisfy the security requirements, users need to:

- Support all features that are classified as "MUST" in RFC 8446 [183] and RFC 6347 [184] for secure communication
- Support Authenticated Encryption with Associated Data (AEAD), specified in Appendix B.4 of RFC 8446 [183] and Section 4.1.2.4 of RFC 6347 [184], that offers both integrity and confidentiality protection
- Support the handshake protocol, defined in Section 4 of RFC 8446 [183] and Section 4.2 of RFC 6347 [184], including the client authentication capability

- Support the per-record nonce to ensure the non-replayability (Section 5.3 in RFC 8446 [183])
- Consider further recommendations according to RFC 7525 [198] and RFC 8996 [147] for a secure TLS implementation

In addition, for the cipher suite, we recommend ChaCha20-Poly1305 [154], for performance and security reasons. As explained in RFC 7905 [114], ChaCha20-Poly1305 is designed to be efficient in software implementations, which outperforms AES-GCM, the only AEAD cipher suite used by all three security protocol candidates. In contrast, AES-GCM has limited performance (see Figure 4.6 and 4.7a) and is not easy for software implementation [114]. For users with hardware accelerations (e.g., HSM, CPU with AES-NI instructions [99]), users should first evaluate cipher suites and select the best performing one. As indicated in § 4.7, HSM on our development board can offer excellent performance gains for AES-128-GCM. Besides, ChaCha20-Poly1305 is proven to be secure [51] and is resilient to side-channel vulnerabilities. Moreover, RFC 7905 [114] and RFC 7634 [153] have stated the support of ChaCha20-Poly1305 for TLS and IPsec, respectively.

Multiple LANs with IPsec. Second, as explained above, IPsec that works at Layer 3 can also provides the end-to-end security for an in-vehicle network with multiple LANs. Besides, IPsec has comparable performance to TLS. To satisfy the security requirements, users need to:

- Support all IPsec features that are classified as "MUST" in RFC 4301 [197] and RFC 4302 [103], except all requirements related to the tunnel mode
- Support the IP Authentication Header (AH) with the transport mode, as stated in RFC 4302 [103]
- Optionally support the IP Encapsulating Security Payload (ESP), as specified in RFC 4303 [104]
- Support the Internet Key Exchange (IKEv2) for the authentication, as described in RFC 7296 [102]

- Support the anti-replay service by verifying the sequence number field in the AH packet header (Section 3.4.3 in RFC 4302 [103])

Note that, IPsec offers security protection for various upper-layer protocols, while TLS or DTLS only protects TCP or UDP traffic, respectively. For example, Internet Control Message Protocol (ICMP) and Internet Group Management Protocol (IGMP) operate directly above Layer 3, which cannot be encapsulated in TLS packets. In this chapter, we consider TLS and IPsec are interchangeable, because they can both protect TCP/UDP traffic. We argue that TCP/UDP traffic in the in-vehicle network is the dominant type of traffic and carries vital control and sensor data; thus, other types of traffic, like ICMP and IGMP, are less security-critical.

Single LAN with MACsec. Third, we consider only a single Local Area Network (LAN) scenario. Although the assumed network architecture in § 4.2.1 involves a gateway that has Layer 3 (L3) routing capabilities, the domain-based in-vehicle network can be configured as a LAN, where the gateway is a Layer 2 (L2) switch instead of an L3 router. In this case, MACsec should be enough to secure the in-vehicle communication. To satisfy the security requirements, users need to:

- Support all mandatory requirements of MACsec and MKA, listed in IEEE 802.1AE-2018 (Annex A) and IEEE 802.1X-2020 (Annex A.9 for MKA only)
- Support the integrity protection using the default cipher suite (i.e., AES-GCM-128) provided by MACsec (Clause 14 in IEEE 802.1AE-2018 [86])
- Optionally support the confidentiality protection using the default cipher suite (Clause 14 in IEEE 802.1AE-2018 [86])
- Support the MACsec Key Agreement protocol (MKA) for the authentication (IEEE 802.1X-2020 [87])
- Support the replay protection by checking the packet number (PN) field in the SecTAG of the received frame (Clause 10.6 in IEEE 802.1AE-2018 [86])

We also have two suggestions on the cipher suite and MKA for MACsec. According to our evaluation results § 4.6.3, the default cipher suite of MACsec, AES-GCM, has the lowest performance. Users can either adopt (1) the hardware-based MACsec or (2) implement other efficient cipher suites. For the former option, hardware chips, like Broadcom BCM82391 [98], can provide considerable performance gain. For the latter option, user-implemented cipher suites should meet the criteria specified in Clause 14.2 of IEEE 802.1AE-2018. For MKA, MACsec can be tuned to have better handshake performance. As mentioned in § 4.6.4, MACsec spends lots of time on MKA, the second stage of its handshake procedure, while the first stage only costs around 337 ms. The low performance of the first stage is due to the large transmission interval, 2 s, for MKA. Fortunately, MACsec supports a shorter MKA interval, 0.5 s (Table 9-3 in IEEE 802.1X-2020 [87]), which can improve its handshake performance.

CHAPTER 5

Rigorous Security Enhancement for CAV System against CV Spoofing Attack

5.1 Introduction

With the emerging Connected Vehicle (CV) technology [211], Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) wireless communication enables vehicles to exchange important safety and mobility information with other entities in real time. While CV technology can significantly benefit transportation mobility and safety, such dramatically increased connectivity inevitably increases the attack surface of CV devices (e.g., vehicles, infrastructures). For example, it is easy for the attacker to send falsified data, interfering with the CV ecosystem. Chen et al. [37] have demonstrated that one single attack vehicle can create massive traffic congestion by sending falsified location and speed data.

Since one malicious sender can reach numerous receivers, it is challenging to ensure that all these receivers have proper and timely protection against spoofing attacks. In a CV environment, the diversity of the receivers (e.g., vehicles, infrastructures, and pedestrians) further increases the challenges. For example, some CV receivers (e.g., pedestrians) may not have the computation power to deploy the anomaly detection system. Thus, to fundamentally solve the problem, it is necessary to prevent data spoofing from the sources: malicious vehicles, even if the system is compromised.

To prevent data spoofing at the vehicle side, our goal is to ensure the integrity of the critical

CV data (e.g., location and speed) from the incoming sensor reading to the outgoing transmission at the On-Board Unit (OBU). However, the challenge is to provide such integrity guarantee in the presence of potential software-layer compromise in in-vehicle systems. In-vehicle systems are known to have a large attack surface, including a broad range of Electronic Control Units (ECUs) (e.g., CD players, Bluetooth, and cellular radio) [34] and more recently malware in the IVI (In-Vehicle Infotainment) system [141]. More seriously, vehicle owners can compromise their own vehicles. Thus, as long as in-vehicle systems are not vulnerability-free, such compromises are always achievable in practice.

To address this challenge, we design and implement CVSHIELD, leveraging recent advances in hardware-assisted security to provide strong security guarantees. More specifically, CVSHIELD uses a hardware feature called Trusted Execution Environment (TEE) (e.g., ARM TrustZone [14]). With this hardware feature, the execution environment is split at the processor level into a normal world and a secure world. The normal world runs a commodity OS, which provides a Rich Execution Environment (REE), and is completely isolated from the secure world running the TEE. Thus, security-critical data and code can be put in the TEE to guarantee their confidentiality and integrity even if the commodity OS is compromised. Such isolation can effectively reduce the attack surface from the whole OS to the code and data residing in TEE, making it much harder to compromise. Also, since the size of the code and data in TEE is much smaller, it is easier to ensure its correctness through formal verification or manual review.

CVSHIELD can protect the pipeline of sensor data (1) reading, (2) processing, (3) encapsulation, and (4) transmission in CV (§5.3). For (1) and (4), CVSHIELD first disables the normal world from accessing peripherals directly, as TEE can control all memory/peripheral accesses and interrupts received by the normal world. Then, CVSHIELD relocates drivers of peripherals into TEE, which include sensors and CV network interface, so that only TEE is capable of interacting with security-critical peripherals. (2) and (3) are often handled by user-space applications. However, the codebases of these applications (e.g., `gpsd` [62]) are usually large, so putting them as a whole into TEE will significantly increase the TCB size. Therefore, CVSHIELD analyzes these appli-

cations only to extract necessary code sections. Since manual extraction of security-critical code sections is laborious and error-prone, we propose to utilize the idea of program slicing [224] to extract code sections of sensor data processing automatically. Apart from protecting the pipeline, CVSHIELD also exposes trusted APIs to the normal world for sensor information reading and CV packet transmission, as other applications in the normal world should be able to retrieve sensor data and exchange CV packets. For example, a trusted GPS API will provide the latest location information. For the overall design, we should not violate the real-time constraints of CV [68], which requires the vehicle to broadcast Basic Safety Messages (BSMs) every 100 ms. Overall, our research goals are summarized as follows:

- Design and implement a TEE-based system CVSHIELD to protect CV sensor data integrity and prevent CV spoofing attack at the vehicle side.
- Leverage program slicing to reduce TCB size and extract code sections on sensor data processing automatically.
- Optimize the overall system performance to meet the real-time requirement of CV.

5.2 Threat Model

In this chapter, we trust the device hardware and exclude the threat of hardware attacks (e.g., GPS spoofing [233]). Both secure and non-secure kernels should be able to load the properties of hardware devices correctly (e.g., physical addresses and buses, interrupts); otherwise, neither kernel can exchange data with sensors.

Following existing works [37, 130], to ensure the functionality of TEE, we assume that the boot ROM and boot loader are trusted so that the secure kernel can be faithfully loaded. On the other hand, the non-secure OS, system services, and all user-space applications in the normal world may be malicious. This is possible, because previous works [34, 108] have already shown that in-vehicle systems can be compromised physically or remotely. Note that attacks that aim at compromising

TCB (i.e., all code that runs in TEE) are out of the scope. As long as in-vehicle systems are not vulnerability-free, such compromises are always possible in practice.

5.3 Design of CVSHIELD

As presented in § 5.1, we aim at protecting sensor data integrity and prevent spoofing attacks at the vehicle side, by relocating code sections of sensor data reading, processing, encapsulation, and transmission into TEE. The following are design goals of CVSHIELD:

1. *Data correctness*: the normal world should not be able to directly access security-critical peripherals (e.g., GPS, CV network interface).

2. *Functionality*: applications in the normal world should be able to access sensor information (e.g., GPS location, vehicle speed) and transmit CV packets.

3. *Low complexity*: irrelevant code sections should be removed from TEE in order to reduce TCB size.

4. *Usability*: code extraction and relocation should be automated and should try to exclude human efforts, which may be laborious and error-prone.

Besides four design goals, we should consider the real-time constraints of the CV network while implementing CVSHIELD.

To achieve Goal (1), we first utilize a TrustZone-specific feature to configure access permissions of security-critical peripherals. In § 2.3, we mention that i.MX products are equipped with a TrustZone-compatible component, CSU. Although CVSHIELD prototype is built on i.MX6 SoC, if other SoCs provide similar security functionality, CVSHIELD is general and can be ported to other SoCs. Then, to allow TEE to exchange data with peripherals, we relocate necessary device drivers into TEE, such as the serial device driver required by GPS. After that, by programming the ARM General Interrupt Controller (GIC), CVSHIELD registers interrupt handlers for protected pe-

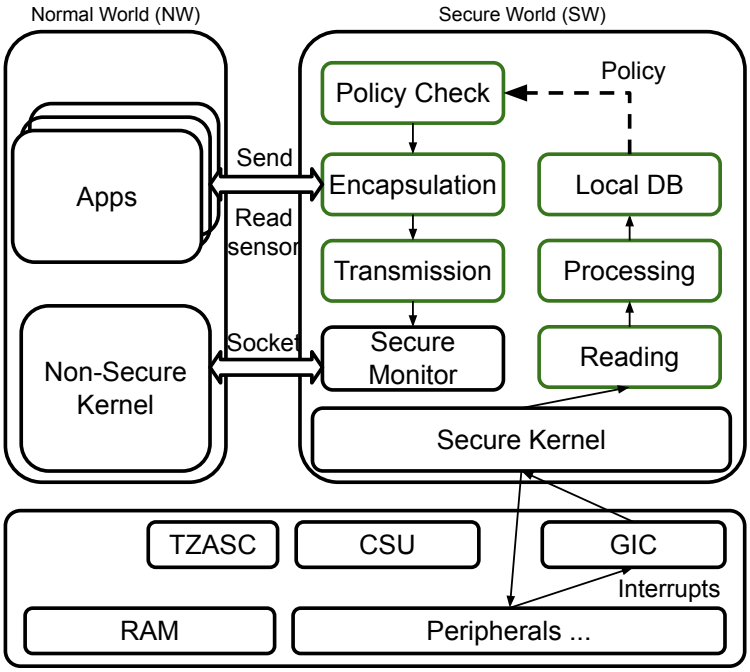


Figure 5.1: CVSHIELD extends the trust boundary to protect code sections related to sensor data in green boxes.

ripherals. When new data arrives at the peripherals, only the secure world can receive the interrupt and retrieve the data.

For Goal (2), we first must port code sections of sensor data processing and encapsulation into TEE. Then, we expose trusted APIs to the normal world, so applications in the normal world can still retrieve sensor information (e.g., location and speed) and transmit CV packets. In this case, CVSHIELD can understand the raw sensor data and extract the sensor information; also, it can validate outgoing CV packets before the transmission. However, unlike device drivers, sensor data processing and encapsulation programs usually have large codebases and contain irrelevant code sections, which may violate Goal (3). For example, `gpsd` [62] is a commonly used daemon that receives data from a GPS receiver and provides the data back to multiple user-space applications. Parsing GPS data and generating location reports is just a small portion of it. Thus, we propose to utilize program slicing [224] to remove irrelevant parts. Static program slicing takes the source code and a slicing criterion (e.g., a call site of some function) as input. It then performs static

analysis to generate a program slice that may affect the values of the slicing criterion. For instance, the potential slicing criterion for `gpsd` can be the function of generating location reports. Also, program slicing can help us reduce human efforts in code extraction, so Goal (4) can be guaranteed.

To ensure the real-time requirement in the CV environment, we break down the time overhead for each component in Figure 5.1 and optimize the performance case by case. In § 5.6, we show our efforts in optimizing sensor data reading, which can eliminate overhead introduced by context switches between the normal and secure worlds.

5.4 CVSHIELD Static Analysis

In this section, we will describe how the program slicing can be used to remove irrelevant code sections in the sensor data processing programs.

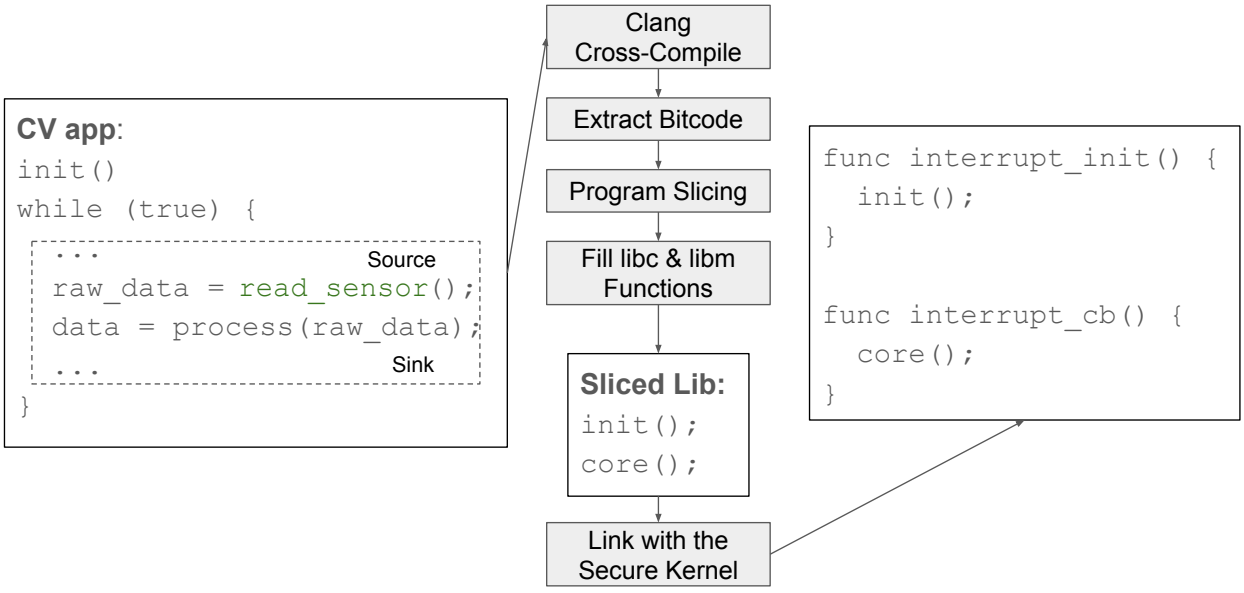


Figure 5.2: The workflow of CVSHIELD static analysis

5.4.1 Sensor Data Processing Program Characterization

We observe that sensor data processing programs often share a similar pattern. Sensor data processing programs first keep reading and parsing raw data from sensors. Then, the parsed data will

be encapsulated into user-defined structures. Based on our observations, we *abstract* the sensor data processing program into two parts:

- **Initialization**: the initialization part contains code sections that start from the program entry to the beginning of the core processing part. Specifically, this part is responsible for initializing the execution context of the core processing part. For example, any local or global variables used in the core processing part must be allocated and initialized. Without the initialization part, the core processing part may not work correctly.

- **Core processing**: the core processing part includes code sections that read and parse raw sensor data. Since the sensor data processing program keeps running, the core processing part is often wrapped in a loop. For each loop iteration, the program will first call an I/O API to obtain the raw sensor data. Then, the raw data will be parsed and encapsulated into a user-defined structure, which will be stored in a variable or transmitted via the I/O API (e.g., network channel).

To identify the two parts described above, CVSHIELD requires the developer to annotate two locations where the raw sensor data is obtained (*source*) and encapsulated (*sink*). For example (Listing 5.1), `gpsd` [62] requires two annotations on (1) `read`, a socket API, and (2) `handler`, a callback function to process the encapsulated GPS data.

5.4.2 Program Dependence Graph

CVSHIELD reasons about the sensor data processing programs using a program dependence graph (PDG) [132, 57, 207], referred to as P. PDG represents each LLVM instruction as a vertex, and the edges capture control and data dependencies between instructions. That is, a PDG is a combination of a control dependence graph (CDG) and a data dependence graph (DDG) for a given program. With the PDG, CVSHIELD can extract a minimal set of instructions that are related to sensor data processing, using program slicing [224, 132].

```
ssize_t packet_get(int fd, struct gps_lexer_t *lexer) {
  ssize_t recvd;
  // sensor data source
  recvd = read(fd, lexer->inbuffer + lexer->inbuflen,
     sizeof(lexer->inbuffer) - (lexer->inbuflen));
  ...
  // parsing the data
  packet_parse(lexer);
  ...
}

gps_mask_t gpsd_poll(struct gps_device_t *session) {
  ssize_t newlen;
  ...
  newlen = packet_get(session->gpsdata.gps_fd, &session->lexer);
  ...
  // encapsulating the parsed data
  gps_merge_fix(&session->gpsdata.fix, session->gpsdata.set,
     &session->newdata);
  ...
}

int gpsd_multipoll(const bool data_ready, struct gps_device_t *device,
    void (*handler)(struct gps_device_t *, gps_mask_t), float
    reawake_time) {
  if (data_ready) {
    // target loop
    while (true) {
      gps_mask_t changed = gpsd_poll(device);
      ...
      if (device->lexer.type != BAD_PACKET) {
        // sensor data sink
        handler(device, changed);
      }
    }
  }
  ...
}
```

Listing 5.1: GPSD annotations

Control Dependence Graph. A control dependence graph (CDG) embeds all control dependencies for a given program. For any two instructions I_1 and I_2, I_2 is *control dependent* on I_1 if the execution of I_2 is determined by the outcome of I_1. For example, the condition value in an branch instruction (i.e., *if-else* statement) can control which instruction will execute next. To build a CDG, the control flow analysis and post-dominator analysis are required. Using the CDG, we can infer the control dependencies between the sensor data *source* and *sink*. However, to extract a complete set of instructions that are only for sensor data processing, the CDG is not enough.

Data Dependence Graph. To infer a complete set of dependencies, a data dependence graph (DDG) is another essential component. For any two instructions I_1 and I_2, I_2 is *data dependent* on I_1 if the correctness of the execution result of I_2 is determined by the outcome of I_1. In other words, I_2 uses the data produced by I_1. To construct a DDG, we need to infer the both *def-use* dependence and *read-after-write* memory dependence among LLVM instructions of the given program [132].

The LLVM framework provides the *def-use* dependence information. However, it is challenging to infer the *read-after-write* memory dependence, which requires *pointer analysis*. However, scaling the pointer analysis to the whole program is challenging. Therefore, we incorporate PtrSplit's parameter-tree approach [132] to avoid the global pointer analysis. We follow KSplit's design [81] to employ SVF's wave-propagation-based Andersen's pointer analysis [202] for the inter-procedural pointer analysis (i.e., for each function). For each caller-callee pair, the parameter-tree approach connects the actual parameter trees for arguments at the call sites with formal parameter trees for parameters in the function definition. Consequently, the intra-procedural pointer analysis results can be propagated inter-procedurally.

5.4.3 Code Extraction

Boundary identification. Before slicing the sensor data processing programs, we need to first identify the boundary between the initialization and the core processing part, which is the loop described in § 5.4.1. Listing 5.1 indicates that the *source* and *sink* may not directly appear in the target loop. Therefore, as specified in Algorithm 4, the loop that we are looking for must be able to

reach both the *source* and *sink*. For this case, we leverage the PDG to query the reachability of any two graph nodes. We iterate over all loops in the program and inspect call instructions in the loop. If the loop can reach both the *source* and *sink*, we will mark the loop as the target loop. Moreover, if both the *source* and *sink* are wrapped in multiple nested loops, we will choose the innermost loop.

Algorithm 4: Identify the boundary between the initialization and the core processing part.

Input: PDG P, Source N_{src}, Sink N_{sink}, LLVM module M
Output: The target loop L_t

```
1  L_t = NULL;
2  foreach loop L in M do
3      canReachSource = false;
4      canReachSink = false;
5      foreach call instruction CI in L do
6          canReachSource| = P.canReach(CI, N_src);
7          canReachSink| = P.canReach(CI, N_sink);
8      end
9      if canReachSource and canReachSink then
10         if L_t == NULL or L_t.contains(L) then
11             L_t = L;
12         end
13     end
14 end
```

Backward slicing. As mentioned above, CVSHIELD extracts the initialization and the core processing using PDG-based program slicing. It's worth noting that we perform an instruction-level slicing as stated in Algorithm 5, instead of the function-level slicing [132]. The slicing algorithm is standard [224, 24]. Algorithm 5 describes a worklist-based algorithm, which begins with the annotated *sink* node. For each node in the worklist, we collect its neighbor nodes that have control/data dependence edges to the node. Note that, these dependence edges are directional, which point to the current node obtained from the worklist. We repeat this procedure until the worklist becomes empty. By doing so, we gather a set of dependence nodes that lead to the *sink* node. Since the PDG is constructed over LLVM instructions, we can infer the corresponding instructions from the set of collected dependence nodes. Besides, the program slicing technique also ensures that the

extracted part is executable, as relevant control and data dependency information are retained.

Algorithm 5: PDG-based backward slicing.

Input: PDG P, Sink N_{sink}
Output: A set of dependence nodes S

```
1  S = {N_sink};
2  worklist = S;
3  while !worklist.empty() do
4  |   n = worklist.pop();
5  |   forall control/data dependence edge from n' to n do
6  |   |   if !S.contain(n') then
7  |   |   |   S.insert(n');
8  |   |   |   worklist.push(n');
9  |   |   end
10 |   end
11 end
```

Next, we use the target loop to divide the set of relevant instructions into the initialization and the core processing part. Given the target loop L_t, we label its parent function as F_t (target function). For the example in Listing 5.1, F_t is `gpsd_multipoll` function. For each instruction inferred from S, if the instruction belongs to the target function F_t, the parent function of L_t, but is not in the loop body of L_t, we will categorize this instruction into the initialization part. Additionally, if the selected instruction does not belong to F_t, there must exist a path from the current instruction to the target function in the PDG. Furthermore, we also assign global variables, which are used in S, into the initialization part. Since the initialization and the core processing part are exclusive in S, the remaining instructions that are not marked as the initialization part belong to the core processing part.

Post-processing. After extracting the initialization/core processing part, we then run a pass to further remove unused code, using a fixed point algorithm (Algorithm 6). We iterate over all functions and global variables in the LLVM module of the program. If the function or the global variable has no user, we will erase it from the module. We repeat this step until the analysis reaches a fixed point where all functions and global variables are alive, except for those in the allowed list. Additionally, we utilize the dead argument elimination pass in LLVM to clean up unused function

arguments or return values. In this way, we can further reduce irrelevant code in the original program.

Besides, since the sensor data processing programs work in the user space, many external functions may not exist in the secure kernel of TEE. Therefore, we collect a list of external functions that are used in the extracted code sections. For example, `libc` and `libm` are commonly used in sensor data processing programs, while the secure kernel does not include all of them, because the secure kernel needs to be small. To ensure the successful execution of the sensor data processing program in the secure kernel, the list of external functions should be manually imported into the secure kernel. Note that, integrating the sensor data processing program and the dependent external functions enlarges the footprint of the secure kernel. CVSHIELD aims to reduce the size of the imported sensor data processing program. The developer should be aware of such a trade-off and leverage other approaches (e.g., program testing, formal methods) to ensure the security of the newly imported code sections.

Algorithm 6: Remove unused code.

Input: LLVM module M, a list of kept functions/global variables $allowlist$

```
1  fixpoint = false;
2  repeat
3      fixpoint = true;
4      foreach function F in M do
5          if !allowlist.contain(F) and F.users().empty() then
6              F.erase();
7              fixpoint = false;
8          end
9      end
10     foreach global variable GV in M do
11         if !allowlist.contain(GV) and GV.users().empty() then
12             GV.erase();
13             fixpoint = false;
14         end
15     end
16 until fixpoint;
```

5.5 Implementation

5.5.1 Sensor Data Reading

As mentioned in § 5.3, we relocate device drivers into the secure kernel. Before that, we integrate the CSU driver and enable the TZASC-380 driver to allow secure-only access for sensor peripherals. We leverage the serial device driver for the GPS data and the network device driver for the CV network data. For GPS I/O, We reuse `imx_uart` driver in the OP-TEE codebase, which provides read/write capabilities of the serial device. Since the serial device transmits data byte by byte, we further wrap it up to provide C-style I/O API. For CV network I/O, we port `libethdrivers` provided by `seL4`'s `util_libs` [193, 194]. In addition, `lwip` is needed to handle low-level network packet headers (e.g., Ethernet frame, IP, and UDP header).

5.5.2 Static Analysis for Sensor Data Processing

Following the proposed design in § 5.4, we implement a set of LLVM passes to perform the static analysis in CVSHIELD (Figure 5.2). The LLVM passes are implemented in 1555 lines of C++, including the boundary identification pass, code extraction pass, post-processing pass, and external function collection pass. For the alias analysis, we follow the existing work [81] and employ the SVF framework [202]. Based on the results of the external function collection pass, we manually integrate required `libc` and `libm` functions into OP-TEE, which are imported from `newlib` [152] (a C library for the embedded system).

5.6 Evaluation

In this section, we first present the TCB size of CVSHIELD. Then, we conduct experiments to evaluate the performance of the current implementation.

5.6.1 Testbed Setup

Our testbed is based on the Boundary Devices Nitrogen6Q development board [28], namely the SABRE Lite board. The overall hardware of this board is similar to commercial OBUs [45]. The board has 1GB of memory and contains an i.MX6 SoC with a quad-core ARM Cortex-A9 processor with TrustZone security extensions. We use Linux kernel version 4.9.128 [27], provided by Boundary Devices, as the non-secure kernel. The secure kernel is based on OP-TEE [125].

5.6.2 TCB Size

Following the existing work [120], we also break down the source lines of code for CVSHIELD. Table 5.1 shows the source lines of code for different sections of the secure kernel implementation. The main contents in the table summarize all components that are added for CVSHIELD. As indicated by its name, the "Core" section provides the most critical kernel functionalities, including memory management, threading, timer, and the secure monitor. For the GPS data, we develop a Pseudo Trusted Application (PTA) that works in the secure world and exposes secure APIs to the normal world.

For the "Drivers" section, we add two device drivers to the secure kernel: the serial device driver (UART) and the network device driver. Apart from these, the secure kernel has already included GIC, TZASC-380, etc. For the "Library" section, `libm` is a dependency of `gpsd`, and `lwip` is imported for the network communication. Note that, `gpsd` in the secure kernel is extracted from the corresponding user-space program. As mentioned in § 5.5.2, the CVSHIELD analysis reports all external functions that are used in `gpsd`, in which many of them are `libm` functions. We will discuss how to further reduce the TCB size in § 5.7.

In summary, our newly introduced components occupy 30% of the code base in the secure kernel. The GPS PTA is the only interface that is exposed to the normal world, which introduces a limited attack surface. The user in the normal world can only read the GPS data.

For the code reduction percentage, we compare the number of LLVM instructions before and after the program slicing. The reduction percentage of `gpsd` is not ideal, only 7%. One reason

Type	C Src	C Hdr	ASM	Total
Core				
GPS PTA	515	45	0	560
Drivers				
Serial device	275	15	0	290
Network	4676	4403	0	9079
Libraries				
libm	20576	2844	0	23420
lwip	47955	11945	0	59900
gpsd	4150	1686	0	5836
Total	78147	20938	0	99085
Others	146833	33049	6797	186679
Total (whole TEE)	224980	53987	6797	285764

Table 5.1: Breakdown of the source lines of code (SLOC) for different components in the secure kernel.

is the over-approximation of the static program analysis that is inherited from PtrSplit [132]. We will discuss this issue later in § 5.7. Another reason is that there are many compound structures in `gpsd`'s code base, which blocks us from inferring a correct and complete set of dependencies. Therefore, we manually simplify the extracted `gpsd` parts. The total number of SLOC is reported in Table 5.1, while the full version of `gpsd` has 48777 SLOC of C code in total.

5.6.3 System Performance Measurement

To understand the performance of sensor reading, we expose `read_s` API to the normal world and measure its performance with the baud rate of 115200. In the normal world, we develop a small test program that calls the exposed API repeatedly. The test program invokes the read function 1000 times for different data sizes. We measure the execution time of each function call and calculate the average value. Table 5.2 summarizes the results on exposed APIs. For a real GPS device, the polling interval is about 0.044 seconds, and the average size of fetched data is around 7.91 bytes. Our current implementation can handle real GPS data and only takes around 0.686 ms to read 7.91 bytes of data for each polling.

We then measure the performance of reading the parsed data. The size of the parsed GPS data

Size (bytes)	10	100	1000	10000
`read_s` (ms)	0.87	8.79	87.98	880.02
`read_mem` (ms)	0.79	8.71	87.91	879.90

Table 5.2: Performance of exposed serial device I/O APIs.

(i.e., a GPS fix) is a constant (192 bytes for `gpsd`). Thanks to the abstraction of the sensor data processing (§ 5.4.1), it only takes 0.042 ms to read the latest GPS location information. For each triggered interrupt on the GPS device, the secure kernel will be triggered to process the newly received data, which runs in the background for the user in the normal world. Thus, the *core processing* part can simultaneously buffer and parse the raw GPS data. Once a GPS packet is recognized, the parsed GPS data will be updated. For each function call, the user directly reads the memory region that stores the parsed GPS data, without waiting until the raw data parsing is finished.

Optimization. We analyze the execution time of the exposed API (see Figure 5.3) and propose a passive communication mechanism based on the shared memory, because we want to reduce introduced overhead as much as possible. In comparison with the normal case (Figure 5.3a), the extra time overhead in Figure 5.3b comes from (1) context switches (around 0.041 ms) and (2) memory copy between the normal world and the secure world. In § 2.3, we mention that the context switch (i.e., `smc` instruction) generates a synchronous exception and suspends the execution in the normal world. To avoid synchronous context switches, we design a passive communication mechanism that utilizes asynchronous hardware interrupts. As shown in Figure 5.3c, the secure kernel handles interrupts from hardware devices using an idle CPU core. Then, the secure kernel writes data into a read-only memory chunk that is shared with the normal world. In this case, the applications do not need to trigger context switches, but only monitor the shared memory (e.g., an array) and process the latest data accordingly. Based on this design, we implement a sensor data reading API `read_mem`. By comparing two rows in Table 5.2, we confirm that our optimization can reduce the time overhead caused by context switches.

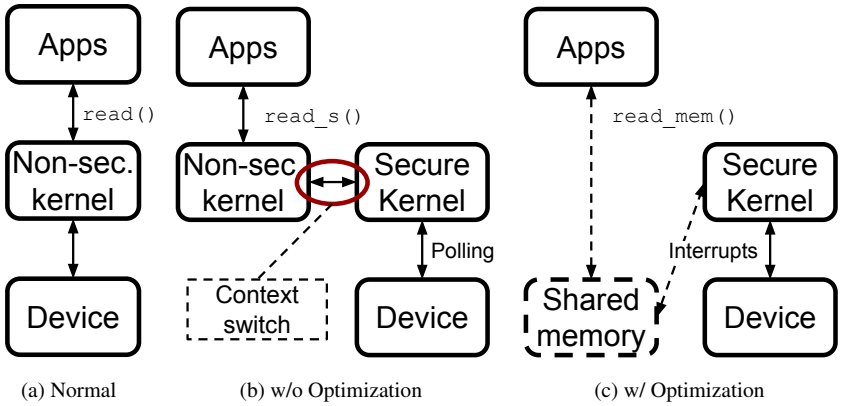


Figure 5.3: Optimization of sensor data reading. (a) does not incorporate TEE, while (b) shows a trusted API of sensor reading. (c) avoids context switches and eliminates the time overhead via shared memory.

5.7 Discussion

Automate the code extraction for device drivers. As mentioned in § 2.5, KSplit [81] extends PtrSplit [132] to isolate device drivers from the Linux kernel. KSplit performs static analysis on the source code of the kernel and the device driver. The analysis is used to identify the shared states between the kernel and the driver. Since the driver will be isolated from the kernel, the shared states between the kernel and the driver must be synchronized correctly. CVSHIELD is complementary to KSplit. We focus on automating the extraction of the user-space sensor data processing parts, while KSplit aims to isolate the kernel-space device driver. Potentially, KSplit can be extended to relocate the device driver to the secure kernel automatically. However, the secure kernel has different services from the Linux kernel, such as memory and thread management. Many APIs used in the source code of the device driver may not exist in the secure kernel. To bridge the gap for the device driver relocation, we may provide a detailed API map between the Linux kernel and the secure kernel so that the device driver can be transformed accordingly.

Dynamic program slicing. The static program analysis is known to be undecidable [113, 71], leading to imprecise approximations. On the other hand, the dynamic program analysis [2] can capture the actual behavior of the program, given a specific set of inputs. CVSHIELD relies on PtrSplit to construct the PDG, while PtrSplit's design choices incur an over-approximation, which

still retains irrelevant code in the extracted slice. For example, indirect function calls (e.g., function pointers) are challenging to handle for CFG construction. PtrSplit uses the function signature in the indirect call instruction to match potential callee functions, forming a superset of the correct ones. That is, the constructed call graph introduces redundant edges, in which the corresponding indirect function calls will never happen in the program behaviors. Also, over-approximation happens in the DDG construction. More specifically, while inferring the *read-after-write* memory dependence, for each load instruction, PtrSplit enumerates all store instructions in the same function and checks if they refer to the same memory locations, using the pointer analysis. However, this design choice does not consider the instruction order, which results in redundant DDG edges. Therefore, in future work, we would like to combine dynamic program slicing to alleviate the over-approximation issue caused by the static program analysis. Besides, software debloating [176, 177, 41], which aims at trimming unused code of a program, can be further incorporated to reduce the size of the relocated device drivers and newly added libraries.

5.8 Conclusion

Rapid advances in CV technology have increased the probability of cyberattacks (e.g., spoofing attacks) [37]. In this chapter, we propose a TEE-based system, CVSHIELD, to protect sensor data integrity. Overall, CVSHIELD relocates security-critical code sections of sensor data reading, processing, encapsulation, and transmission into TEE so that the malicious attacker in the normal world cannot modify and transmit falsified data. To automate the security-critical code extraction and minimize TCB size, we utilize program slicing to remove code sections that are irrelevant to sensor data processing. To ensure the functionalities of normal-world applications, we expose trusted services to provide the latest sensor information and CV packet transmission capability with the normal world. Also, we consider optimization during the design and development phases to ensure the efficiency of CVSHIELD.

CHAPTER 6

Vulnerability Discovery of CAV System under Physical-World Attacks

6.1 Introduction

After years of research and testing, the Connected and Autonomous Vehicle (CAV) technology is now used to assist human driving with automated functionalities. Due to its safety-critical nature, CAV manufacturers must ensure that the system functionalities should not be interrupted by malicious attackers. However, the powerful Connected and Autonomous Vehicle (CAV) system brings new security challenges to vehicular systems, because newly introduced communication and system modules inevitably increase the attack surface of vehicles. Security researchers have demonstrated that physical-world attacks can affect different modules of the CAV system, especially ML-related modules [33, 32, 199, 188, 77].

Moreover, the security of the CAV system itself is largely under explored [230, 201], especially under physical attacks. The CAV system introduces various modules and many library dependencies, which raise the bar of the security analysis. Notably, compromising the CAV system may jeopardize its normal functioning, leading to performance degradation and system crashes. Thus, we should thoroughly analyze the CAV system itself.

Fuzzing is a practical and efficient dynamic analysis approach for vulnerability discovery, which has been widely used in both the academia [15, 189, 190, 26, 58] and industry [143, 61]. On the other hand, the dynamic analysis approach, like fuzzing, incurs low overhead while testing

real-world systems. That is, the scalability of fuzzing is much better than other analysis/testing approaches (e.g., model checking, static program analysis, symbolic execution). Therefore, we propose to conduct the fuzzing study on the CAV system and present CAVFUZZER, aiming at uncovering semantic vulnerabilities that can be triggered on the road. Nonetheless, it is still challenging to apply fuzzing to the CAV system, due to the cyber-physical nature of the CAV system.

(C1) First, identified vulnerabilities are not exploitable/triggerable from the physical world without considering physical constraints. The classic mutation strategies corrupt the program inputs at the byte level, which is meaningless for the CAV system. Even the grammar-based mutator [15] will not help in the vulnerability discovery of the CAV system, because the semantic information of the surrounding environment is essential while constructing meaningful fuzzing inputs. For example, the attacker cannot place the sticker or 3D object above the ground.

(C2) Second, the coverage feedback is ineffective for detecting semantic vulnerabilities. Machine learning (ML) models play essential roles in the CAV system. However, the execution of ML models includes mainly numeric operations [230, 166]. For different inputs, ML programs produce almost the same code coverage results. Our initial results also validate this observation. The fuzzer will not consider input as interesting, if the input results in the same code coverage as previous inputs. Therefore, many inputs will be ignored by the fuzzer, even if these inputs lead to wrong decision results.

(C3) Third, the performance of fuzzing the CAV system is fairly low, due to the complexity of the CAV system. For example, Baidu Apollo [9] involves more than 10 functionality modules, such as perception, localization, control, and planning. The program under testing has to be reset to the initial state for each fuzzing input. Given the internal complexity of the CAV system, resetting the CAV system inevitably introduces extra time overhead. Besides, existing research works often use a simulator (e.g., LGSVL [121], CARLA [206]) while testing the CAV system [122, 79, 238, 237]. Combining the simulator and the CAV system further slows down the performance of the fuzzing. For instance, AV-FUZZER [122] takes an average of 3.6 minutes to evaluate each driving scenario.

Therefore, to address (C1), we prototype an object-level mutator for sensor inputs of the CAV

perception module, which also considers practical physical-world attack capabilities [30, 151, 138]. Specifically, the object-level mutator first attaches physical objects to the sensor inputs. Then, the mutator will apply semantic mutation strategies to place new objects or modify existing objects. For instance, primitive mutation operations for camera inputs include but are not limited to blurring and rotation. With the proposed object-level mutator, we can ensure that the generated objects upon the sensor inputs can be reproduced in the physical world.

For (C2), we combine data-flow feedback with the control-flow coverage to assist the analysis of the CAV system. ML is an essential part of the CAV system; therefore, we propose incorporating the data-flow feedback to guide the fuzzer for ML-based CAV systems. One well-known metric for testing the neuron network is the neuron coverage [166]. However, our exploration indicates that the neuron coverage is not efficient in fuzzing the CAV system. Besides, we further define domain-specific objectives (e.g., emergency stop function) for the CAV system for semantic vulnerabilities.

Regarding (C3), we adopt manually constructed fuzzing harnesses and avoid using the simulator to improve the fuzzing efficiency. As mentioned in § 2.4, the CAV system consists of multiple functional modules. Usually, a runtime framework is needed to coordinate these modules, such as Cyber RT [10] and ROS [160]. The runtime framework is not reachable from the physical world and is independent of functional modules. Therefore, it is unnecessary to include the runtime framework in the fuzzing phase. To exclude the runtime framework, we manually construct a fuzzing harness for the perception module that can be executed independently. Notably, we do not modify the core processing logic of the perception module. To avoid using the simulator, we can add an offline phase to record data traces (e.g., sensor data) in the simulator. Then, during the fuzzing phase, we can replay the recorded traces and mutate the input sensor data. In this case, we can reduce the overhead introduced by the simulator at the runtime.

Overall, our key contributions are summarized as follows:

- We design and prototype CAVFUZZER, aiming at uncovering semantic vulnerabilities (e.g., wrong perception results) that can be triggered on the road.
- We present an object-level mutator for Camera inputs of the CAV system and introduce the

loss-based data-flow feedback to guide the fuzzing.

- We demonstrate that CAVFUZZER can uncover semantic vulnerabilities efficiently and effectively by comparing the number of identified vulnerability instances with a neuron-coverage-guided fuzzer [166].

6.2 Threat Model

Since we consider physical-world attacks, we should embed the *practical* attacker's capabilities into the security analysis. We assume the attacker resides in the physical world and is close to the victim CAV. Besides, we allow the attacker to place the adversarial patch [30] or 3D object [32] in the surrounding location of the victim CAV. The split-second phantom attack is also within the scope, in which the attacker can use a portable projector or a digital billboard to embed new objects in the camera sensor. The ideas of these physical-world behaviors are borrowed from existing works [30, 32, 151, 138], because they can affect the behavior of the CAV system and can be controlled by a physical-world attacker. Meanwhile, the attacker aims to trigger the semantic vulnerability that can lead to wrong perception results in the CAV system. Such attacks can further mislead the downstream modules, like prediction and planning modules.

6.3 Design and Implementation of CAVFUZZER

Figure 6.1 presents the high-level workflow of CAVFUZZER, introducing three new components (i.e., green boxes in the figure) to assist the fuzzing. We first present the high-level design of CAVFUZZER in § 6.3.1. Then, we describe the details of the object-level mutator in § 6.3.2 and the data-flow feedback/objective in § 6.3.3.

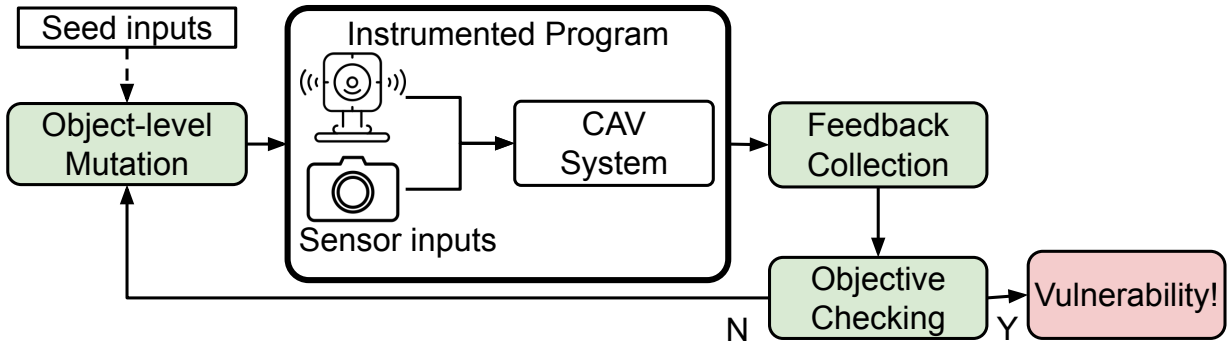


Figure 6.1: Overview of CAVFUZZER

6.3.1 High-Level Design

CAVFUZZER is designed to uncover semantic vulnerabilities in the CAV system, specifically the perception module. Since the perception module is the first component in the processing pipeline of the CAV system, any incorrect results in the perception module can affect other downstream modules. Therefore, we are interested in understanding how malicious sensor inputs can mislead the perception module to generate wrong prediction results.

Existing fuzzing approaches, like AFL [143], have certain limitations for our analysis goal. First, the classic fuzzing tools are designed to test command-line programs. For our case, the CAV system obtains the input data from sensor devices. Therefore, we need to interface the fuzzer with the semantic inputs from the sensor. Second, as mentioned in § 6.1, the byte-level mutation is meaningless for the CAV system. For the perception module, the high-level goal is to identify obstacles in the surrounding environment. Minor byte-level modifications on the sensor inputs may not change the semantic information (i.e., the perception results). On the other hand, extensive corruption of the sensor inputs may result in wrong perception results, but it is impractical and not reproducible in the physical world. Third, the control-flow coverage feedback does not work well in detecting semantic vulnerabilities. Due to the use of ML models in the CAV system [230, 166], exercising the program with different inputs produces almost the same code coverage results.

To achieve our analysis goal, we design CAVFUZZER, an extensible fuzzing tool for the CAV system, which introduces the object-level mutation and data-flow information to assist the fuzzing.

CAVFUZZER is built upon LibAFL [59], a framework to build modular and reusable fuzzers. Currently, we implement the object-level mutation for the camera input and develop loss-based feedback to guide the fuzzer.

6.3.2 Object-Level Mutation

As the first step in the workflow of CAVFUZZER, we apply the object-level mutation to the seed inputs (i.e., benign sensor inputs). The goal of the mutation is to generate diverse mutated inputs such that the system under testing can output unexpected results (e.g., wrong perception results).

According to the threat model (§ 6.2), the attacker can place objects in the physical world to add objects to the sensor inputs. In our tool, we choose to attach "patches" to the sensor inputs. Such a design choice is based on existing works about adversarial patches [30], which have been shown to be effective in fooling image classifiers. Usually, the adversarial patch is generated for a specific ML model, and it is not easy for the fuzzer to produce a perfect patch from scratch (e.g., from a purely white patch). Therefore, we collect a set of existing patches to bootstrap the fuzzing process. Although the collected patches may not directly mislead the machine model in the system under testing, we believe that applying image-specific mutations on collected patches can produce usable ones. This design decision is also based on the seed selection work [72] in the fuzzing community, highlighting the importance of the initial seeds in bootstrapping the fuzzer.

Our mutator starts with a benign camera input (e.g., an image) and a set of existing patches. At the initialization stage, CAVFUZZER loads clean images into a queue and existing patches into a set. The fuzzer then picks an image from the queue for each fuzzing round and enters the mutation stage. Instead of mutating the whole image, we mutate the patch and overlay it to the chosen image. Specifically, we define three types of mutation strategies:

- **Primitive mutation**: this mutation strategy aims at modifying the presence and meta properties (i.e., size, location) of objects, including four operations: (1) add, (2) delete, (3) resize, and (4) move. The first two operations are complementary. They randomly insert a loaded patch or remove an inserted patch that is attached to the image. For any inserted patch, the

fuzzer may resize the patch or change its location in the image.

- **Local mutation**: the local mutation operates on a partial region of the patch. The included mutation operations are inherited from AFL's havoc mutations, which modify one or consecutive bytes of the raw data of the patch. We exclude any havoc mutations that change the raw data size; instead, we rely on the resize operation in the primitive mutation to change the patch size.

- **Global mutation**: this type of mutation globally modifies the patch. Although local mutations can aggressively modify the patch, it may take longer for the patch to evolve to show observable changes. Therefore, we introduce global mutations, including two primary operations: (1) blur and (2) add noise. These two operations are complementary as well. The blur operation reduces the edge content and smooths the color transition in the patch, which is helpful for removing noise. On the other hand, the noise addition operation introduces extra random data to the patch. The current CAVFUZZER prototype utilizes OpenCV [161] to implement the global mutation, including averaging blurring, Gaussian blurring, median blurring, bilateral filtering blurring, and Gaussian noise.

Note that, we do not directly overlay mutated patches with the chosen image until we feed the mutated image to the system under testing. That is, every image in the queue carries a list of mutated patches. The list is empty for the clean image. For each patch in the list, we record the patch's location, size, and raw data so that we know how to merge mutated patches with the underlying image input.

6.3.3 Objective and Data-Flow Feedback

After obtaining the mutated sensor input, we feed it to the system under testing and monitor the collected feedback and execution results. In addition to the coverage-guided fuzzing, we instrument the system to obtain the data-flow feedback, specifically loss-based feedback for the ML model in the system under testing.

Objective. Our goal is to uncover semantic vulnerabilities, like wrong perception results. We define the fuzzing objective as follows. For each image input, we evaluate the corresponding execution results against the ground truth. The ground truth results are collected while exercising the system with the *clean* image. A mutated image input is considered *vulnerable*, if any of the following three conditions are met:

- The detection probability of any existing objects in the ground truth is decreased;
- New detected objects are introduced to the results;
- Any existing objects in the ground truth are removed from the results.

Data-flow feedback. For the coverage-guided fuzzing, the program under testing is instrumented at the compile time (e.g., AFL [143, 58, 59]). Besides, hitcounts of each edge are also logged to a shared bitmap during the execution, in which each byte represents an edge [58]. To avoid path explosion, the hitcount is bucketed to a power of two. However, the perception module in the CAV system highly depends on the ML model, which mainly includes numeric operations [230, 166]. For different inputs, the ML-based system produces almost the same code coverage results, as shown in Figure 6.2.

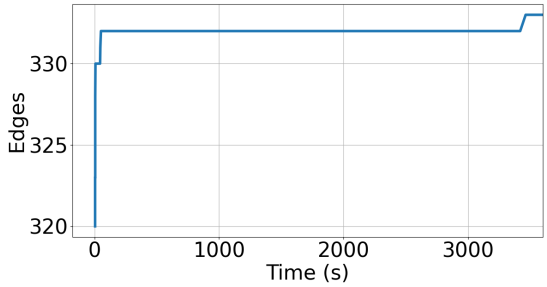


Figure 6.2: Edge coverage over time while fuzzing an ML-based perception module.

According to the fuzzing objective defined above, we would like to see that the new execution results of the mutated image can largely deviate from the ground truth. That is, we need a metric that can quantitatively measure the difference between the new execution results and the ground

truth. Thus, we choose to use the loss function of the ML model, as it measures how far an estimated value is from its ground-truth value, which satisfies our requirements. The loss function is not fixed and varies case by case. In our prototype, we reuse the loss function of YOLOv4 [25], as Baidu Apollo [9] also uses YOLOv4 in its camera perception module. Table 6.1 presents the loss values among 5 different images. The first column and the first row describe the main object in the image. The values in the diagonal cells of the table are always the minimum value in the corresponding row. We denote the loss value between two identical images as the *ground-truth loss* value. The mutated image is considered *interesting*, if the loss value between the mutated image and the clean one is larger than the ground-truth loss.

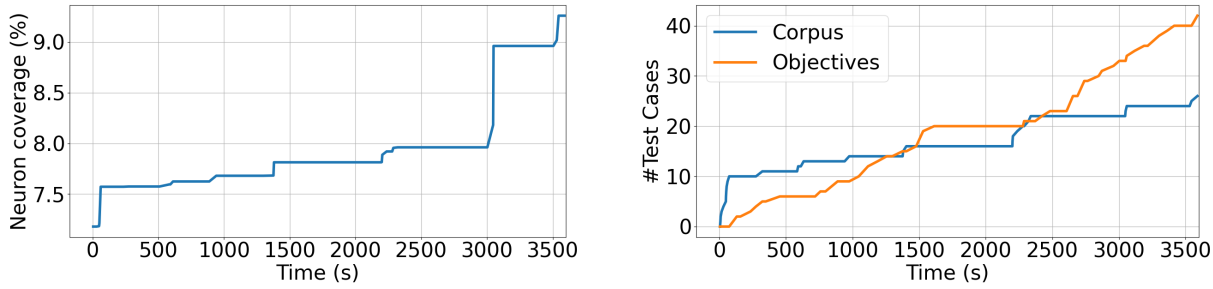
	dog	eagle	giraffe	horses	person
dog	**1.034905**	11.924377	15.891864	19.622555	19.031891
eagle	6.217122	**0.000494**	5.954041	9.931243	10.347088
giraffe	9.315841	7.262587	**0.002287**	12.716213	13.063067
horses	17.998217	15.882026	17.525015	**0.198861**	19.23213
person	12.487648	10.693492	11.531662	15.277912	**0.001878**

Table 6.1: The loss values among 5 different images.

Before compiling the system under testing, we manually instrument the ML-based system to collect the results from the output layer of the ML model, which are stored in a shared memory region and will be used in the loss calculation. At the initialization stage, we calculate the ground-truth loss value and ground-truth execution results for the clean image. Later, for each fuzzing round, we will record the output layer results and the execution results. By comparing with the ground truth, CAVFUZZER reports *vulnerable* mutated images to the user and saves *interesting* ones to the queue for future mutations.

6.4 Evaluation

We evaluate the efficiency and effectiveness of CAVFUZZER on a Ubuntu 20.04 server with two 8-core Intel Xeon Silver 4110 2.10GHz CPUs, 96 GB memory, and four NVIDIA GeForce RTX



(a) Neuron coverage over time

(b) Produced interesting and vulnerable inputs over time

Figure 6.3: Neuron-coverage-guided fuzzing for 1h.

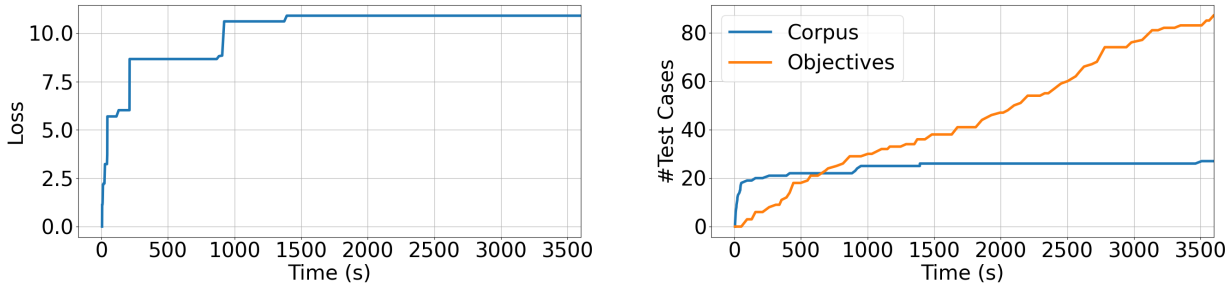
2080 GPUs. We prototype CAVFUZZER for the camera perception module (using YOLOv4 [25]) of Baidu Apollo, which is a mature and open-source AV platform.

6.4.1 CAVFUZZER Efficiency and Effectiveness

To validate the efficiency and effectiveness of CAVFUZZER, we first run two experiments to compare with the neuron coverage [166]. For both of them, we stick with the same objective as defined in § 6.3.3, but configure the data-flow feedback differently:

- **Object mutation + neuron coverage**: for this experiment, we enable all three categories of object-level mutations and the neuron coverage feedback [166].
- **Object mutation + loss feedback**: this configuration is the full version of the current CAVFUZZER prototype. We replace the neuron coverage with our proposed loss-based feedback.

Figure 6.3 and Figure 6.4 present the results for two experiments. In the right figures, the blue line represents the number of *interesting* test cases in the corpus queue, which push towards the increase of the data-flow feedback value (see § 6.3.3). The orange line illustrates the trend of the total number of *vulnerable* over time. By comparing Figure 6.3b and 6.4b, we observe that our loss-guided fuzzing can generate vulnerable image inputs 2x faster than the neuron-coverage-guided fuzzing, which validates the efficiency of CAVFUZZER. Meanwhile, by inspecting the percep-



(a) Loss value over time

(b) Produced interesting and vulnerable inputs over time

Figure 6.4: Loss-guided fuzzing for 1h.

tion results of the collected *vulnerable* inputs, we can confirm that they all introduce unexpected results (e.g., newly detected objects), which demonstrates the effectiveness of finding semantic vulnerabilities in the CAV system.

6.4.2 Case Studies

Figure 6.5 illustrates two concrete cases while feeding the generated vulnerable inputs to Baidu Apollo's camera perception module. In Figure 6.5a, except for the original detection results, a van is detected with very high confidence (99.1%), while its bounding box overlays with the actual car. Since the newly detected van is "parked" on the side of the road and does not block the road, this vulnerable input is less likely to affect the driving behavior of the ego vehicle. For the other case, as shown in Figure 6.5b, three pedestrians are detected in the front of the ego vehicle, and their detection confidence is larger than 90%. Although the ego vehicle drives in an urban area at a low speed, the newly detected pedestrians may still trigger an emergency stop on the ego vehicle, as the distance between them is pretty small.

(a) New van (99.1%)

(b) Three new pedestrians (>90%)

Figure 6.5: New objects are inserted to the perception results of Baidu Apollo's perception module.

6.5 Discussion and Future Work

Mutation practicality. In the current prototype of CAVFUZZER, we attach "patches" to the seed sensor inputs and semantically mutate their contents. We assume the attacker can place the patch in the physical world by projecting a digital version or printing a physical one. However, it is not easy to ensure that the projector or the printer faithfully retains all details of the generated patches (e.g., color, angle, light, pattern). Therefore, we should consider more physical constraints in the mutator for the camera input mutation. We admit that the practicality of our current mutator is limited. To improve the mutation, we can borrow the idea of Expectation Over Transformation (EOT) [16] from the ML community. EOT is used to generate robust adversarial examples. Since EOT models 3D rendering and printing, the generated adversarial examples can mislead image classifiers over various angles, viewpoints, and lighting conditions. For our object-level mutator, while applying existing mutation strategies (e.g., resize, blur, rotate) to the patch, we can further add minor perturbations (e.g., angle, viewpoint, and lighting changes) to the generated patch. We will obtain a set of mutated patches and treat the set as one single input. Only if the whole set of inputs satisfies the objective, can we assert that one instance of the malicious input is identified. By doing so, we should be able to improve the practicality of the object-level mutator.

Mutation generality. At the high level, the proposed three categories of object-level mutation strategies are general, but the mutation operations in the CAVFUZZER prototype are only appli-

cable to the image inputs. One direction of improving CAVFUZZER is to support more types of inputs for the CAV system. We can explore the domain-specific mutations for other sensors, such as LiDAR, radar, and even the CV network interface. Taking LiDAR as an example, which is the key enabler for the CAV technology, placing 3D objects around the CAV [32] is a promising threat model for attacking the LiDAR perception system. The object-level mutation can then modify various properties (e.g., size, location, shape) of the 3D object.

General and automated data-flow instrumentation. Recent advances in the fuzzing community have explored the use of data-flow information to guide the fuzzer. DDFuzz [139] utilizes the data dependency graph (DDG) to assist the fuzzing. Apart from the code coverage, the fuzzer considers an input interesting if any new edges in the DDG are hit. An LLVM pass for DDG instrumentation is shipped with DDFuzz, which can be applied during the compile time. The authors of DDFuzz observe that fuzzing the data dependency edges does not help to increase the code coverage but can assist the fuzzer to "stress" the already explored code locations thoroughly. datAFLow [73] is another recent work that introduces data-flow coverage. The authors believe the data flow can more accurately characterize program behaviors and uncover more bugs than the code coverage. As a future direction, besides the loss-based feedback in CAVFUZZER, we can incorporate the general data-flow information to guide the fuzzer.

Snapshot fuzzing. As mentioned above, we would like to reduce the overhead caused by resetting the CAV system. To achieve this goal, snapshot fuzzing can be further adopted. Snapshot fuzzing has been used for fuzzing hypervisors [189] and network applications [190]. In short, the fuzzer first obtains a snapshot (e.g., memory state) of the system's state directly before executing the fuzzing input. After each fuzzing run, we can reset the system to a deterministic state by replacing the snapshot. The cost of this reset is independent of startup complexity and only determined by the size of the changes to the state of the system caused by executing the fuzzing input [190].

6.6 Conclusion

In this chapter, we design CAVFUZZER, an extensible fuzzing solution for the CAV system to uncover semantic vulnerabilities. Specifically, CAVFUZZER introduces an object-level mutator that considers the physical-world attack capabilities. Also, CAVFUZZER utilizes loss-based feedback to guide the fuzzer, instead of the control-flow coverage. We demonstrate that CAVFUZZER can efficiently identify semantic vulnerabilities, compared with a neuron-coverage-guided fuzzer.

CHAPTER 7

Future Work and Conclusion

This chapter discusses potential future research directions and concludes the dissertation.

7.1 Future Work

7.1.1 Remote Attacks against CAVs

The CV network enables multiple vehicles to work cooperatively, which will be a more valuable target for the attacker, when the CV network becomes popular in the future. Among the attack surfaces discussed in the dissertation, CAV attacks against the in-vehicle network and the CAV sensors require the attacker to be present at the surrounding locations of the victim vehicle. In contrast, the CV network communication interface has the potential for large-scale remote attacks. First, more and more collaborative applications that are built upon CV communication are proposed recently (e.g, collaborative perception [236, 179, 178, 36, 35, 222, 129, 180], cooperative driving automation in the CARMA platform). Second, the attacker can even launch attacks against CAVs through the Internet, because, eventually, the CV network will connect the infrastructures with the vehicles and provide Internet connectivity. The attacker can exploit newly emerging CV applications to launch remote attacks and affect multiple CAVs in a large-scale way.

7.1.2 Defense against CAV Threats via Cross-Validating Different CAV Inputs

Many attack works [77, 33, 32, 37, 188, 199, 205] against the CAVs have been proposed in recent years. However, a systematic defense against existing CAV threats is missing. Prior works often target one attack surfaces (e.g., CV network [77, 37], sensor inputs [33, 188, 199, 205]). Intuitively, it is difficult for the attacker to simultaneously attack all three CAV system inputs (i.e., attack surfaces). Therefore, the inconsistency among different CAV system inputs (i.e., CV network, in-vehicle network, and sensor inputs) can help detect attacks. Users can develop an anomaly detection system to inspect three types of data for inconsistency. The anomaly detection can be executed regarding different properties, such as vehicle status, detected objects, and time-series consistency. Besides, users can leverage information from other vehicles to assist in anomaly detection. For example, the object reshaping attack [32] generates a 3D-printed object that the perception module cannot detect. Nonetheless, the object reshaping attack is optimized for a specific model in a fixed viewpoint. If any surrounding vehicles can detect the 3D-printed object and report it through CV communication, such a threat may be thwarted.

7.1.3 Verification of the Safety-Critical CAV System

Due to the safety-critical nature of the CAV system, its security guarantee should never be ignored. Formal verification would be a powerful approach to verify the correctness and security guarantee of the CAV system [228, 234]. In Chapter 3 and 4, we have demonstrated that model checking approaches and the Tamarin prover can help us uncover critical design flaws and ensure the security guarantee of the proposed defense. However, we only study a small piece of the whole CAV system, while other modules are under-explored. As mentioned in previous chapters, the CAV system consists of many functional modules. Zhang et al. [234] propose AVChecker to verify if the CAV system driving rules can follow the traffic rules in different scenarios. Apart from the traffic rule checking, another aspect that cannot be ignored is the real-time property of the CAV

system. For example, the V2X module of the CAV system (i.e., the CV network interface) should be able to report its location and speed information every 100 ms. Also, the control command must be issued in a timely manner while facing complex road conditions. To systematically verify the real-time property of the CAV system, we need to construct a formal computation model for the CAV system and verify if a task can be scheduled efficiently to be finished with a specific time bound. The worst-case execution-time (WCET) analysis may be a potential choice for this line of future work.

7.2 Conclusion Remarks

After years of research and testing, we can envision that, in the future, the CAV will be a widely-used and essential technology in our daily work and life. Nowadays, as the CAV system is getting more and more complex, introducing new attack surfaces, it becomes difficult to assure the expected security of the CAV system. Such urgent security needs further stimulate researchers to thoroughly analyze the CAV system and provide strong security guarantees, before the large-scale deployment.

This dissertation proposes practical solutions to understand design- and implementation-level flaws in the current CAV system. Also, we inspect the corresponding security/safety consequence of identified issues so that we can enhance the CAV system accordingly. Specifically, utilizing formal methods, program analysis, and the trusted execution environment (TEE), my dissertation research systematically (1) detect design-level flaws in CV communication protocols, (2) design practical broadcast authentication approach for the next-generation in-vehicle Ethernet network, (3) present rigorous security enhancement against CV spoofing attack, and (4) uncover semantic vulnerabilities against the CAV system. In summary, this dissertation demonstrates that: Proactive vulnerability discovery and security enhancement of the CAV system can (1) uncover new security vulnerabilities, (2) systematically examine fundamental vulnerability causes and security consequences, and (3) provide a strong security guarantee for the defense mechanisms.

APPENDIX A

Summary of More CV Attacks

N2 *Request Mute Attack*: This attack injects a malicious learning response with the same `HashedId8` value of `ca1`. Thus, `V2` chooses to remove the matching entry with the `HashedId8` value of `ca1`. `V2` fails in sending a learning request because `V2` wrongly thinks she has learned the unknown certificate but not.

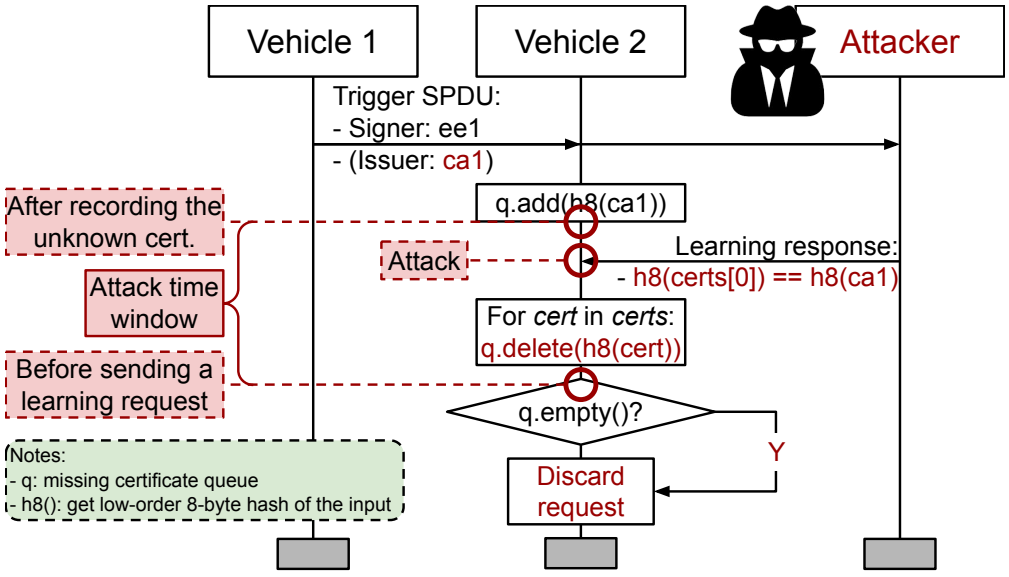


Figure A.1: *N2*: the attacker can stop V2 from sending learning requests to V1 by sending a malicious learning requests.

Assumptions. Similar to *N1*, we assume that `V1` does not mandate the verification for incoming learning responses. Also, we assume that the attacker has enough computing power to efficiently construct a learning response that can cause partial hash collision (e.g., low-order 8 bytes collision).

Attack steps. As shown in Figure A.1, V2 initializes P2PCD after receiving a trigger SPDU from V1. V2 stores the HashedId8 value of the unknown certificate ca1 in a queue. Meanwhile, since the attacker can observe the trigger SPDU, she constructs a malicious learning response, in which the HashedId8 value of the first certificate in the payload matches with the unknown certificate ca1. As defined in P2PCD, after receiving a learning response, V2 extracts all certificates in the learning response and stores them via AddCertificate. At this time, V2 wrongly thinks that it has successfully learned the unknown certificate but actually not. Thus, V2 removes the entry of the unknown certificate h8(ca1) in the queue, where h8 is a function to get the low-order eight-byte hash of the input. As the queue becomes empty, V2 decides not to attach the learning request in the next outgoing SPDU. Consequently, V2 is unable to learn the correct unknown certificate.

Discussion. Similar to *N1*, *N2* is also caused by the use of truncated hash, and the attacker does not need to possess a legitimate certificate. In IEEE 1609.2, the issuer field in a certificate is a HashedId8 value. Therefore, on receiving the trigger SPDU, the vehicle can only store the truncated hash value in the queue. This opens a door for the partial hash collision attack. Although HashedId8 is larger than HashedId3 and makes the attacker harder to find a hash collision, a resourceful attacker (e.g., nation-states, terrorists) can always have enough computing power to efficiently find the hash collision. The attacker can even prepare these malicious learning responses in an offline way. On the other hand, due to the optional verfication of the learning response, it is still possible that some poorly implemented CV protocols may not verify the incoming learning response but just store them.

A4 *Split Trigger Attack*

Assumptions. We assume that the leader keeps the configurations (e.g., platoon size, members) hidden from followers [8].

Attack steps. *A4* requires the attacker to be the leader of the victim platoon, which is consist of V1 and V2 sequentially. After becoming the leader, the attacker immediately sends a SPLIT_REQ to V1. At the last step of the split maneuver, the attacker sends a SPLIT_DONE to V1, which contains necessary platoon configuration data. Notably, the attacker can control the optimal platoon size in

SPLIT_DONE and sets it to 1. Since V1, as a follower, does not store any platoon configurations, it can only trust the attacker. However, the platoon size exceeds the optimal platoon size; V1 thus initiates the split maneuver. Most importantly, *A4* leads to a chain reaction that V1 will pass the wrong configuration to the *last* member in the victim platoon (i.e., V2 in this case). Moreover, V1 and V2 will not able to merge into other platoons or accept any incoming merge requests, because there is no available space.

A5, A6 *Merge Disruption Attack*: The attacker initiates a merge maneuver but does not faithfully complete the whole procedure, so the victim platoon leader V1 is trapped at the busy state and cannot switch back to the idle state. Therefore, V1 cannot process any incoming messages.

Attack steps. In *A5*, the attacker first sends a MERGE_REQ to V1. Since there exists available space in the victim platoon, V1 will accept the request and send a MERGE_ACCEPT to the attacker. In the normal case, V1 will wait for a MERGE_DONE from the merge request initiator. However, the attacker chooses not to send a MERGE_DONE; thus, V1 will keep waiting.

In *A6*, the attacker first utilizes *A2* to join the victim platoon. If V1 initiates a merge maneuver to join a front platoon and receives a MERGE_ACCEPT, V1 will inform all the followers, including the attacker, to change their platoon leader by sending CHANGE_PL to them. The attacker can either passively wait for the happening of the merge maneuver or intentionally trigger the merge maneuver of V1 by conducting *the platoon takeover attack* (*A1*). As a malicious follower of V1, after receiving a CHANGE_PL from V1, the attacker chooses not to reply with an ACK. According to the merge FSM in [8], V1 will keep sending CHANGE_PL to the attacker if V1 does not receive the corresponding ACK.

A8-9 *Split Disruption Attack*: *A8* and *A9* have the same goal and consequence as *A5* and *A6*, but have different attack targets. They focus on vulnerabilities of the split maneuver.

Attack steps. In *A8* and *A9*, the attacker first joins the platoon, which is consist of V1 (leader) and V2 sequentially, by launching *A2* and acts as a malicious follower. In *A8*, V1 sends a SPLIT_REQ to the attacker. After accepting the request, the attacker does not respond to the following CHANGE_PL sent by V1. Therefore, V1 will not be able to switch back to the idle state.

Differently, in *A9*, V1 sends a SPLIT_REQ to V2, the splitting vehicle. After V2 accepting the split request and acknowledging CHANGE_PL, V1 needs to inform the follower behind the attacker to change the platoon leader. The attacker can remain silent, keeping both V1 and V2 at the busy state.

A10 *Follower Block Attack*: This attack is the immediate consequence of *A1* and is more powerful than *A5-9*, because this attack can block all vehicles in the victim platoon rather than one or two of them. All members in the victim platoon will be unable to respond any incoming platoon messages.

Attack steps. The attacker first takes over the victim platoon. Then, she sends SPLIT_REQ to all her followers (i.e., V1 and V2). V1 and V2 accept the split request and reply with SPLIT_ACCEPT. Following the protocol, the attacker sends CHANGE_PL to V1 and V2. After that, the attacker can drive away or keep silence; all followers thus will never receive SPLIT_DONE from the attacker and keep sending ACK.

A11 *Gap Attack*: The basic idea of this idea is to prevent the vehicle from "creating" enough space in the front of the splitting vehicle during the leader/follower leave maneuver.

Attack steps. The attacker is the last follower in the victim platoon and initiates a follower leave maneuver. V1 approves the leave request sent by the attacker. Then, the attacker faithfully respond to SPLIT_REQ and CHANGE_PL from V1. To make the attacker a free agent, V1 sends a SPLIT_DONE to the attacker. Before the completion of the leave maneuver, V1 has to guarantee that there exists enough space at the front of the attacker to perform lane change. If the attacker does not send a GAP_CREATED, V1 will keep busy as it wrongly thinks the leave maneuver is still on-going.

A12, A13 *Leave Disruption Attack*: *A12* and *A13* exploit timers in the leader leave maneuver and the follower leave maneuver respectively.

Attack steps. In *A12*, when the leader wants to leave the platoon, its followers have to elect a new leader. The elected leader then sends a ELECTED_LEADER to the old leader who then hands over the leadership to the elected leader, by initiating the leader leave maneuver and safely leave

the platoon. However, if the attacker is one of the followers (*A2*) and becomes the elected leader, she can choose not to respond. As well, *A10* can be used to mislead all followers to a busy state in advance, so no followers can send `ELECTED_LEADER` to the leader, blocking the leader leave maneuver.

In *A13*, a follower wants to leave the platoon and sends a `LEAVE_REQ` to the leader; if no response is received from the leader, the follower is unable to finish the follower leave maneuver. The attacker can place herself at the position of the leader through *A1*, and keep silent. On the other hand, the attacker can utilize *A5-9* to prevent the benign leader from communicating with other followers. Thus, the victim follower cannot finish the follower leave maneuver.

A14 *Dissolve Disruption Attack*: To make a follower unavailable, the attacker can either use *A10* to block all followers or join the victim platoon as a silent follower through *A2*.

BIBLIOGRAPHY

[1] Ahmed Abdo, Sakib Md. Bin Malek, Zhiyun Qian, Qi Zhu, Matthew Barth, and Nael B. Abu-Ghazaleh. Application level attacks on connected vehicle protocols. In *Proc. RAID*, 2019.

[2] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *Proc. ACM PLDI*, 1990.

[3] F Ahmed-Zaid, F Bai, S Bai, C Basnayake, B Bellur, S Brovold, G Brown, L Caminiti, et al. Vehicle safety communications–applications (vsc-a) final report. Technical report, 2011.

[4] F Ahmed-Zaid, F Bai, S Bai, C Basnayake, B Bellur, S Brovold, G Brown, L Caminiti, et al. Vehicle Safety Communications–Applications (VSC-A) Final Report: Appendix Volume 1 System Design and Objective Test. Technical report, 2011.

[5] F Ahmed-Zaid, F Bai, S Bai, C Basnayake, B Bellur, S Brovold, G Brown, L Caminiti, et al. Vehicle Safety Communications–Applications (VSC-A) Final Report: Appendix Volume 3 Security. Technical report, 2011.

[6] Devdatta Akhawe, Prateek Saxena, and Dawn Song. Privilege separation in {HTML5} applications. In *Proc. USENIX Security*, 2012.

[7] Emad Aliwa, Omer Rana, Charith Perera, and Peter Burnap. Cyberattacks and countermeasures for in-vehicle networks. *ACM Comput. Surv.*, 2021.

[8] Mani Amoozadeh, Hui Deng, Chen-Nee Chuah, H. Michael Zhang, and Dipak Ghosal. Platoon management with cooperative adaptive cruise control enabled by VANET. *Vehicular Communications*, 2015.

[9] Baidu Apollo. An open autonomous driving platform. https://github.com/ApolloAuto/apollo, 2022.

[10] Baidu Apollo. ROS (Robot Operating System). https://cyber-rt.readthedocs.io/en/latest/, 2022.

[11] Krzysztof R. Apt and Dexter Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 1986.

[12] ARM Ltd. Address Space Controllers – Arm. https://tinyurl.com/uxjdnfz, 2019.

[13] ARM Ltd. SMC Calling Convention - ARM Infocenter. https://tinyurl.com/y6gkrpo3, 2019.

[14] ARM Ltd. TrustZone – Arm Developer. https://tinyurl.com/vj29ybd, 2019.

[15] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: fishing for deep bugs with grammars. In *Proc. NDSS*, 2019.

[16] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples. In *Proc. ICML*, 2018.

[17] AUTOSAR. Specification of secure onboard communication. *AUTOSAR CP R19-11*, 2019.

[18] Fan Bai and Hariharan Krishnan. Reliability analysis of DSRC wireless communication for vehicle safety applications. In *IEEE ITSC*, 2006.

[19] Elaine Barker, Allen Roginsky, and Richard Davis. Recommendation for cryptographic key generation (revision 2). 2020.

[20] David A. Basin, Cas Cremers, and Catherine A. Meadows. Model checking security protocols. In *Handbook of Model Checking*. 2018.

[21] David A. Basin, Jannik Dreier, Lucca Hirschi, Sasa Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proc. ACM CCS*, 2018.

[22] John Bellardo and Stefan Savage. 802.11 denial-of-service attacks: Real vulnerabilities and practical solutions. In *Proc USENIX Security*, 2003.

[23] Vincent Bénony. Hopper. https://www.hopperapp.com/, 2019.

[24] David W Binkley and Keith Brian Gallagher. Program slicing. *Advances in computers*, 1996.

[25] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[26] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proc. ACM CCS*, 2017.

[27] Boundary Devices. boundarydevices/linux-imx6. https://tinyurl.com/w66kfkj, 2019.

[28] Boundary Devices. i.MX6 ARM Development Board. https://boundarydevices.com/product/bd-sl-i-mx6/, 2019.

[29] Benedikt Brecht, Dean Therriault, André Weimerskirch, William Whyte, Virendra Kumar, Thorsten Hehn, and Roy Goudy. A security credential management system for V2X communications. *IEEE Trans. Intelligent Transportation Systems*, 2018.

[30] Tom B. Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. *CoRR*, abs/1712.09665, 2017.

[31] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. USENIX Security*, 2004.

[32] Yulong Cao, Ningfei Wang, Chaowei Xiao, Dawei Yang, Jin Fang, Ruigang Yang, Qi Alfred Chen, Mingyan Liu, and Bo Li. Invisible for both camera and lidar: Security of multi-sensor fusion based perception in autonomous driving under physical-world attacks. In *Proc. IEEE S&P*, 2021.

[33] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z. Morley Mao. Adversarial sensor attack on lidar-based perception in autonomous driving. In *Proc. ACM CCS*, 2019.

[34] 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 *Proc. USENIX Security*, 2011.

[35] Qi Chen, Xu Ma, Sihai Tang, Jingda Guo, Qing Yang, and Song Fu. F-Cooper: Feature based cooperative perception for autonomous vehicle edge computing system using 3d point clouds. In *Proc. ACM/IEEE SEC*, 2019.

[36] Qi Chen, Sihai Tang, Qing Yang, and Song Fu. Cooper: Cooperative perception for connected autonomous vehicles based on 3d point clouds. In *Proc. ICDCS*, 2019.

[37] Qi Alfred Chen, Yucheng Yin, Yiheng Feng, Z. Morley Mao, and Henry X. Liu. Exposing congestion attack on emerging connected vehicle based traffic signal control. In *Proc. NDSS*, 2018.

[38] Qi Alfred Chen, Yucheng Yin, Yiheng Feng, Zhuoqing Morley Mao, and Henry Xianghong Liu. Vulnerability of Traffic Control System Under Cyber-Attacks Using Falsified Data. In *Transportation Research Board 2018 Annual Meeting (TRB)*, 2018.

[39] Kyong-Tak Cho and Kang G. Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *Proc. USENIX Security*, 2016.

[40] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review*, 2007.

[41] Jake Christensen, Ionut Mugurel Anghel, Rob Taglang, Mihai Chiroiu, and Radu Sion. {DECAF}: Automatic, adaptive de-bloating and hardening of {COTS} firmware. In *Proc. USENIX Security*, 2020.

[42] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An open-source tool for symbolic model checking. In *Proc. CAV*, 2002.

[43] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics*, 2001.

[44] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, 2011.

[45] Cohda Wireless. Mk5 obu. https://tinyurl.com/y6qepj6h, 2019.

[46] Cross-Cutting Technical Committee. Dedicated short range communications (dsrc) message set dictionary™ set. *SAE International*, Mar. 2016.

[47] Cas Cremers and Martin Dehnel-Wild. Component-based formal analysis of 5g-aka: Channel assumptions and session confusion. In *Proc. NDSS*, 2019.

[48] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proc. ACM CCS*, 2017.

[49] Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. Automated analysis and verification of TLS 1.3: 0-rtt, resumption and delayed authentication. In *Proc. IEEE S&P*, 2016.

[50] Cas Cremers, Benjamin Kiesl, and Niklas Medinger. A formal analysis of IEEE 802.11's WPA2: countering the kracks caused by cracking the counters. In Srdjan Capkun and Franziska Roesner, editors, *Proc. USENIX Security*, 2020.

[51] Jean Paul Degabriele, Jérôme Govinden, Felix Günther, and Kenneth G. Paterson. The security of chacha20-poly1305 in the multi-user setting. In *Proc. ACM CCS*, 2021.

[52] David L. Dill. The mur*phi* verification system. In *Proc. CAV*, 1996.

[53] Martin Eian and Stig Fr. Mjølsnes. A formal analysis of IEEE 802.11w deadlock vulnerabilities. In *Proc. IEEE INFOCOM*, 2012.

[54] Jeremy Erickson, Shibo Chen, Melisa Savich, Shengtuo Hu, and Z. Morley Mao. Commpact: Evaluating the feasibility of autonomous vehicle contracts. In *Proc. IEEE VNC*, 2018.

[55] European Telecommunications Standards Institute. Telecommunications and Internet Protocol Harmonization Over Networks (TIPHON) Release 4; Protocol Framework Definition; Methods and Protocols for Security; Part 1: Threat Analysis. *Technical Specification ETSI*, 2003.

[56] Wu-chang Feng. The case for TCP/IP puzzles. In *Proc. SIGCOMM Workshop on FDNA*, 2003.

[57] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1987.

[58] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *Proc. WOOT*, 2020.

[59] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proc. ACM CCS*, 2022.

[60] GlobalPlatform. GlobalPlatform Homepage - GlobalPlatform. https://globalplatform.org/, 2019.

[61] Google. Oss-fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz, 2022.

[62] GPSD project. biiont/gpsd. https://github.com/biiont/gpsd, 2019.

[63] George Gross, Brian Weis, and Dragan Ignjatic. Multicast Extensions to the Security Architecture for the Internet Protocol. RFC 5374, November 2008.

[64] Bogdan Groza, Pal-Stefan Murvay, Anthony Van Herrewege, and Ingrid Verbauwhede. Libra-can: Lightweight broadcast authentication for controller area networks. *ACM Trans. Embed. Comput. Syst.*, 2017.

[65] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proc. MobiSys*, 2017.

[66] Liwei Guo and Felix Xiaozhu Lin. Minimum viable device drivers for ARM trustzone. In *Proc. EuroSys*, 2022.

[67] Peter Hank, Steffen Müller, Ovidiu Vermesan, and Jeroen Van den Keybus. Automotive ethernet: in-vehicle networking and smart mobility. In Enrico Macii, editor, *Proc. DATE*, 2013.

[68] John Harding, Gregory Powell, Rebecca Yoon, Joshua Fikentscher, Charlene Doyle, Dana Sade, Mike Lukuc, Jim Simons, and Jing Wang. Vehicle-to-Vehicle Communications: Readiness of V2V Technology for Application. Technical report, 2014.

[69] Hugh Harney, Andrea Colegrove, Uri Meth, and George Gross. GSAKMP: Group Secure Association Key Management Protocol. RFC 4535, June 2006.

[70] Changhua He and John C. Mitchell. Analysis of the 802.11i 4-way handshake. In *Proc. WiSec*, 2004.

[71] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proc. ACM CCS*, 2018.

[72] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. Seed selection for successful fuzzing. In *Proc. ISSTA*, 2021.

[73] Adrian Herrera, Mathias Payer, and Antony L Hosking. dataflow: Towards a data-flow-guided fuzzer. In *Proc. FUZZING*, 2022.

[74] Gerard J. Holzmann. The model checker SPIN. *Trans. Software Eng.*, 1997.

[75] Hsu-Chun Hsiao, Ahren Studer, Chen Chen, Adrian Perrig, Fan Bai, Bhargav Bellur, and Aravind Iyer. Flooding-resilient broadcast authentication for VANETs. In *Proc. MobiCom*, 2011.

[76] Shengtuo Hu, Qi Alfred Chen, Jiwon Joung, Can Carlak, Yiheng Feng, Z. Morley Mao, and Henry X. Liu. Cvshield: Guarding sensor data in connected vehicle with trusted execution environment. In *Proc. AutoSec@CODASPY*, 2020.

[77] Shengtuo Hu, Qi Alfred Chen, Jiachen Sun, Yiheng Feng, Z. Morley Mao, and Henry X. Liu. Automated discovery of denial-of-service vulnerabilities in connected vehicle protocols. In *Proc. USENIX Security*, 2021.

[78] Yih-Chun Hu, Adrian Perrig, and David B Johnson. Packet leashes: A defense against wormhole attacks in wireless networks. In *Proc. INFOCOM*, 2003.

[79] Zhisheng Hu, Shengjian Guo, Zhenyu Zhong, and Kang Li. Coverage-based scene fuzzing for virtual autonomous driving testing. *CoRR*, abs/2106.00873, 2021.

[80] Lifeng Huang, Chengying Gao, Yuyin Zhou, Cihang Xie, Alan L. Yuille, Changqing Zou, and Ning Liu. Universal physical camouflage attacks on object detectors. In *Proc. IEEE/CVF CVPR*, 2020.

[81] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. {KSplit}: Automating device driver isolation. In *Proc. USENIX OSDI*, 2022.

[82] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. Lteinspector: A systematic approach for adversarial testing of 4g lte. In *Proc NDSS*, 2018.

[83] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol. In *Proc. ACM CCS*, 2019.

[84] IEEE. Ieee standard for a transport protocol for time-sensitive applications in bridged local area networks. *IEEE Std 1722-2016 (Revision of IEEE Std 1722-2011)*, 2016.

[85] IEEE. Iso/iec/ieee international standard - information technology – telecommunications and information exchange between systems – local and metropolitan area networks – specific requirements – part 1ba: Audio video bridging (avb) systems. *ISO/IEC/IEEE 8802-1BA First edition 2016-10-15*, 2016.

[86] IEEE. Ieee standard for local and metropolitan area networks-media access control (mac) security. *IEEE Std 802.1AE-2018 (Revision of IEEE Std 802.1AE-2006)*, 2018.

[87] IEEE. Ieee standard for local and metropolitan area networks–port-based network access control. *IEEE Std 802.1X-2020 (Revision of IEEE Std 802.1X-2010 Incorporating IEEE Std 802.1Xbx-2014 and IEEE Std 802.1Xck-2018)*, 2020.

[88] IEEE. Ieee standard for local and metropolitan area networks–timing and synchronization for time-sensitive applications. *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)*, 2020.

[89] IEEE. P802.1dg – tsn profile for automotive in-vehicle ethernet communications. https://1.ieee802.org/tsn/802-1dg/, 2020.

[90] IEEE 1609 Working Group. Ieee standard for wireless access in vehicular environments (wave) - networking services. *IEEE Std 1609.3-2010 (Revision of IEEE Std 1609.3-2007)*, 2010.

[91] IEEE 1609 Working Group. IEEE Standard for Wireless Access in Vehicular Environments–Security Services for Applications and Management Messages. *IEEE Std 1609.2-2016 (Revision of IEEE Std 1609.2-2013)*, 2016.

[92] IEEE 1609 Working Group. Ieee standard for wireless access in vehicular environments (wave) – multi-channel operation. *IEEE Std 1609.4-2016 (Revision of IEEE Std 1609.4-2010)*, 2016.

[93] IEEE 1609 Working Group. Ieee standard for wireless access in vehicular environments (wave) – networking services. *IEEE Std 1609.3-2016 (Revision of IEEE Std 1609.3-2010)*, 2016.

[94] IEEE 1609 Working Group. 1609 WG - DSRC Working Group. https://tinyurl.com/y2qju2t5, 2017.

[95] IEEE 1609 Working Group. Ieee guide for wireless access in vehicular environments (wave) architecture. *IEEE Std 1609.0-2019 (Revision of IEEE Std 1609.0-2013)*, 2019.

[96] IEEE 802.11 Working Group. Ieee standard for information technology– local and metropolitan area networks– specific requirements– part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications amendment 6: Wireless access in vehicular environments. *IEEE Std 802.11p-2010 (Amendment to IEEE Std 802.11-2007)*, 2010.

[97] IEEE 802.11 Working Group. Ieee standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007)*, 2012.

[98] Broadcom Inc. Bcm82391 28-nm dual 100g macsec retimer phy. https://tinyurl.com/3z2pdsaa, 2022.

[99] Intel. Intel© advanced encryption standard instructions (aes-ni). https://tinyurl.com/2vfr8u3b, 2012.

[100] Ixia. Automotive ethernet: An overview. https://tinyurl.com/ysahfdbj, 2014.

[101] Aris Jules and John Brainard. Client-puzzles: a cryptographic defense against connection depletion. In *Proc. NDSS*, 1999.

[102] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, October 2014.

[103] Stephen Kent. IP Authentication Header. RFC 4302, December 2005.

[104] Stephen Kent. IP Encapsulating Security Payload (ESP). RFC 4303, December 2005.

[105] Sye Loong Keoh, Sandeep Kumar, Oscar Garcia-Morchon, Esko Dijk, and Akbar Rahman. DTLS-based Multicast Security in Constrained Environments. Internet-Draft draft-keoh-dice-multicast-security-08, July 2014. Work in Progress.

[106] Jun Young Kim, Ralph Holz, Wen Hu, and Sanjay Jha. Automated analysis of secure internet of things protocols. In *Proc. ACSAC*, 2017.

[107] Tiffany Hyun-Jin Kim, Cristina Basescu, Limin Jia, Soo Bum Lee, Yih-Chun Hu, and Adrian Perrig. Lightweight source authentication and path validation. In *Proc. ACM SIGCOMM*, 2014.

[108] 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 *Proc. IEEE S&P*, 2010.

[109] Stefan Krauß. Towards a unified view of microscopic traffic flow theories. *IFAC Proceedings Volumes*, 1997.

[110] Hariharan Krishnan and Andre Weimerskirch. "verify-on-demand"-a practical and scalable approach for broadcast authentication in vehicle-to-vehicle communication. *SAE International Journal of Passenger Cars-Mechanical Systems*, 2011.

[111] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. CAV*, 2011.

[112] Leslie Lamport. Real time is really simple. *Microsoft Research*, 2005.

[113] William Landi. Undecidability of static analysis. *LOPLAS*, 1992.

[114] Adam Langley, Wan-Teh Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905, June 2016.

[115] Jan Lastinec and Ladislav Hudec. A performance analysis of IPSec/AH protocol for automotive environment. In Boris Rachev and Angel Smrikarov, editors, *Proc. CompSysTech*, 2015.

[116] Jan Lastinec and Ladislav Hudec. A study of securing in-vehicle communication using ipsec protocol. *Journal of Electrical Engineering*, 2021.

[117] Christine Laurendeau and Michel Barbeau. Threats to security in DSRC/WAVE. In *Proc. ADHOC-NOW*, 2006.

[118] Timm Lauser, Daniel Zelle, and Christoph Krauß. Security analysis of automotive protocols. In Björn Brücher, Oliver Wasenmüller, Mario Fritz, Hans-Joachim Hof, and Christoph Krauß, editors, *Proc. CSCS*, 2020.

[119] Youngwoo Lee and KyoungSoo Park. Meeting the real-time constraints with standard ethernet in an in-vehicle network. In *Proc. IEEE IV*, 2013.

[120] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: ARM trustzone-based mobile peripheral control. In *Proc. MobiSys*, 2018.

[121] LGSVL. Svl simulator by lg. https://www.svlsimulator.com/, 2022.

[122] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael B. Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. AV-FUZZER: finding safety violations in autonomous driving systems. In *Proc. IEEE ISSRE*, 2020.

[123] Hyung-Taek Lim, Kay Weckemann, and Daniel Herrscher. Performance study of an in-car switched ethernet network without prioritization. In *Proc. Nets4Cars/Nets4Trains*, 2011.

[124] Maggie Lim. Automotive Ethernet: The Future of In-Vehicle Networking. https://blogs.keysight.com/blogs/tech/sim-des.entry.html/2021/06/10/automotive_ethernet-E6FB.html, 2021.

[125] Linaro Ltd. OP-TEE/optee_os. https://tinyurl.com/rurhgcl, 2019.

[126] Linaro Ltd. Open Portable Trusted Execution Environment - OP-TEE. https://www.op-tee.org/, 2019.

[127] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *Proc. USENIX ATC*, 2017.

[128] Dongtao Liu and Landon P. Cox. Veriui: attested login for mobile devices. In *Proc. HotMobile*, 2014.

[129] Hansi Liu, Pengfei Ren, Shubham Jain, Mohannad Murad, Marco Gruteser, and Fan Bai. Fusioneye: Perception sharing for connected vehicles and its bandwidth-accuracy trade-offs. In *Proc. SECON*, 2019.

[130] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. Software abstractions for trusted sensors. In *Proc. MobiSys*, 2012.

[131] Jiafa Liu, Di Ma, André Weimerskirch, and Haojin Zhu. Secure and Safe Automated Vehicle Platooning. *IEEE Reliability Society*, 2016.

[132] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proc. ACM CCS*, 2017.

[133] Xin Liu, Ang Li, Xiaowei Yang, and David Wetherall. Passport: Secure and adoptable source authentication. In *Proc. USENIX NSDI*, 2008.

[134] Roger Lucas. DTLS Multicast. Internet-Draft draft-lucas-dtls-multicast-00, September 2017. Work in Progress.

[135] Miao Ma. Mitigating denial of service attacks with password puzzles. In *Proc. ITCC*, 2005.

[136] Hani Mahmassani, Hesham Rakha, Elliot Hubbard, Dan Lukasik, et al. Concept development and needs identification for intelligent network flow optimization (inflo) : assessment of relevant prior and ongoing research. Technical report, 2012.

[137] Hani Mahmassani, Hesham Rakha, Elliot Hubbard, Dan Lukasik, et al. Concept development and needs identification for intelligent network flow optimization (inflo) : concept of operations. Technical report, 2012.

[138] Yanmao Man, Ming Li, and Ryan Gerdes. {GhostImage}: Remote perception attacks against camera-based image classification systems. In *Proc. RAID*, 2020.

[139] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. Fuzzing with data dependency information. In *Proc. IEEE EuroS&P*, 2022.

[140] Kirsten Matheus and Thomas Königseder. *Chapter 6: Ethernet in Automotive System Development*, page 241–263. Cambridge University Press, 2 edition, 2017.

[141] Sahar Mazloom, Mohammad Rezaeirad, Aaron Hunter, and Damon McCoy. A security analysis of an in-vehicle infotainment and app platform. In *USENIX WOOT*, 2016.

[142] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Proc. CAV*, 2013.

[143] Michal Zalewski. american fuzzy lop (2.52b). https://lcamtuf.coredump.cx/afl/, 2022.

[144] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

[145] Tanmaya Mishra, Thidapat Chantem, and Ryan M. Gerdes. Teecheck: Securing intra-vehicular communication using trusted execution. In *Proc. RTNS*, 2020.

[146] John C. Mitchell, Vitaly Shmatikov, and Ulrich Stern. Finite-state analysis of SSL 3.0. In *Proc. USENIX Security*, 1998.

[147] Kathleen Moriarty and Stephen Farrell. Deprecating TLS 1.0 and TLS 1.1. RFC 8996, March 2021.

[148] Lama Moukahal, Mohammad Zulkernine, and Martin Soukup. Vulnerability-oriented fuzz testing for connected autonomous vehicle systems. *IEEE Trans. Reliab.*, 2021.

[149] Silja Mäki, Tuomas Aura, and Maarit Hietalahti. Robust membership management for ad-hoc groups. 2000.

[150] Prasad Narayana, Ruiming Chen, Yao Zhao, Yan Chen, Zhi Fu, and Hai Zhou. Automatic vulnerability checking of ieee 802.16 WiMAX protocols through TLA+. In *Proc. IEEE Workshop on NPSec*, 2006.

[151] Ben Nassi, Yisroel Mirsky, Dudi Nassi, Raz Ben-Netanel, Oleg Drokin, and Yuval Elovici. Phantom of the adas: Securing advanced driver-assistance systems from split-second phantom attacks. In *Proc. ACM CCS*, 2020.

[152] newlib. bminor/newlib. https://github.com/bminor/newlib, 2022.

[153] Yoav Nir. ChaCha20, Poly1305, and Their Use in the Internet Key Exchange Protocol (IKE) and IPsec. RFC 7634, August 2015.

[154] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 8439, June 2018.

[155] NIST NSRC. Benchmarking round 2 candidates on microcontrollers. https://tinyurl.com/vy8udwm4, 2021.

[156] NIST NSRC. Lightweight cryptography — nsrc. https://tinyurl.com/5c3wtwxw, 2021.

[157] OMNeT++. Omnet++ simulator. https://omnetpp.org/, 2020.

[158] OnBoard Security. Aerolink secure vehicle communication. https://tinyurl.com/yaklyx47, 2019.

[159] OnBoard Security. Savari and onboard security partner to bring the most secure v2x solutions to the market. https://tinyurl.com/yy9lzmcu, 2019.

[160] Open Robotics. Cyber RT Documents. https://www.ros.org/, 2022.

[161] OpenCV team. OpenCV. https://opencv.org/, 2022.

[162] OpenSSL. Openssl. https://www.openssl.org/, 2019.

[163] Heejin Park and Felix Xiaozhu Lin. Safe and practical gpu acceleration in trustzone. *arXiv preprint arXiv:2111.03065*, 2021.

[164] Heejin Park and Felix Xiaozhu Lin. Gpureplay: a 50-kb gpu stack for client ml. In *Proc. ASPLOS*, 2022.

[165] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. {StreamBox-TZ}: Secure stream analytics at the edge with {TrustZone}. In *Proc. USENIX ATC*, 2019.

[166] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proc. SOSP*, 2017.

[167] Adrian Perrig, Ran Canetti, Dawn Song, Professor Doug Tygar, and Bob Briscoe. Timed Efficient Stream Loss-Tolerant Authentication (TESLA): Multicast Source Authentication Transform Introduction. RFC 4082, June 2005.

[168] Adrian Perrig, Ran Canetti, Dawn Xiaodong Song, and J. D. Tygar. Efficient and secure source authentication for multicast. In *Proc. NDSS*, 2001.

[169] Adrian Perrig, Ran Canetti, J. D. Tygar, and Dawn Xiaodong Song. Efficient authentication and signing of multicast streams over lossy channels. In *Proc. IEEE S&P*, 2000.

[170] Adrian Perrig, Ran Canetti, J Doug Tygar, and Dawn Song. The tesla broadcast authentication protocol. *Rsa Cryptobytes*, 2002.

[171] Mert D Pesé, Jay W Schauer, Junhui Li, and Kang G Shin. S2-can: Sufficiently secure controller area network. In *Proc. ACSAC*, 2021.

[172] Jonathan Petit, Florian Schaub, Michael Feiri, and Frank Kargl. Pseudonym schemes in vehicular networks: A survey. *IEEE Comm. Surveys & Tutorials*, 2015.

[173] Jonathan Petit and Steven E. Shladover. Potential cyberattacks on automated vehicles. *IEEE Trans. Intelligent Transportation Systems*, 2015.

[174] PLEXE. The platooning extension for veins. plexe.car2x.org, 2019.

[175] Jeroen Ploeg, Bart T. M. Scheepers, Ellen van Nunen, Nathan van de Wouw, and Henk Nijmeijer. Design and experimental evaluation of cooperative adaptive cruise control. In *Proc. ITSC*, 2011.

[176] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. {RAZOR}: A framework for post-deployment software debloating. In *Proc. USENIX Security*, 2019.

[177] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. Slimium: debloating the chromium browser with feature subsetting. In *Proc. ACM CCS*, 2020.

[178] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. AVR: Augmented vehicular reality. In *Proc. MobiSys*, 2018.

[179] Hang Qiu, Po-Han Huang, Namo Asavisanu, Xiaochen Liu, Konstantinos Psounis, and Ramesh Govindan. Autocast: scalable infrastructure-less cooperative perception for distributed collaborative driving. In *Proc. MobiSys*, 2022.

[180] Andreas Rauch, Felix Klanner, Ralph Rasshofer, and Klaus Dietmayer. Car2x-based perception in a high-level fusion architecture for cooperative perception systems. In *Proc. IV*, 2012.

[181] Michael K. Reiter, Kenneth P. Birman, and Li Gong. Integrating security in a group oriented distributed system. In *Proc. IEEE S&P*, 1992.

[182] Stefan Resch and Michael Paulitsch. Using TLA+ in the development of a safety-critical fault-tolerant middleware. In *Proc. ISSRE*, 2017.

[183] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[184] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, January 2012.

[185] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[186] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proc. IEEE/ACM ICSE*, 2016.

[187] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM trustzone to build a trusted language runtime for mobile applications. In *Proc. ASPLOS*, 2014.

[188] Takami Sato, Junjie Shen, Ningfei Wang, Yunhan Jia, Xue Lin, and Qi Alfred Chen. Dirty road can attack: Security of deep learning based automated lane centering under physical-world attack. In *Proc. USENIX Security*, 2021.

[189] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *Proc. USENIX Security*, 2021.

[190] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: Network fuzzing with incremental snapshots. *CoRR*, abs/2111.03013, 2021.

[191] Security Innovation. Security innovation's aerolink software to secure cadillac vehicle-to-vehicle communications. https://tinyurl.com/yy7uacun, 2019.

[192] Michele Segata, Stefan Joerer, Bastian Bloessl, Christoph Sommer, Falko Dressler, and Renato Lo Cigno. Plexe: A platooning extension for veins. In *Proc. VNC*, 2014.

[193] seL4. seL4/util_libs. https://github.com/seL4/util_libs, 2022.

[194] seL4. The seL4© Microkernel. http://sel4.systems/, 2022.

[195] NXP Semiconductors. Mpc5748g microcontroller data sheet. https://tinyurl.com/sna8mm4h, 2018.

[196] NXP Semiconductors. Mpc-ls-vnp-rdb fact sheet. https://www.nxp.com/docs/en/fact-sheet/MPCLSVNPRDBFS.pdf, 2019.

[197] Karen Seo and Stephen Kent. Security Architecture for the Internet Protocol. RFC 4301, December 2005.

[198] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7525, May 2015.

[199] Junjie Shen, Jun Yeon Won, Zeyuan Chen, and Qi Alfred Chen. Drift with devil: Security of multi-sensor fusion based localization in high-level autonomous driving under GPS spoofing. In *Proc. USENIX Security*, 2020.

[200] Christoph Sommer, Reinhard German, and Falko Dressler. Bidirectionally coupled network and road traffic simulation for improved IVC analysis. *IEEE Trans. Mob. Comput.*, 2011.

[201] Rock Stevens, Octavian Suciu, Andrew Ruef, Sanghyun Hong, Michael W. Hicks, and Tudor Dumitras. Summoning demons: The pursuit of exploitable bugs in machine learning. *CoRR*, abs/1701.04739, 2017.

[202] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proc. CC*, 2016.

[203] SUMO. Simulation of Urban MObility. https://sumo.dlr.de, 2020.

[204] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proc. ACM CCS*, 2015.

[205] Jiachen Sun, Yulong Cao, Qi Alfred Chen, and Z Morley Mao. Towards robust {LiDAR-based} perception in autonomous driving: General black-box adversarial sensor attack and countermeasures. In *Proc. USENIX Security*, 2020.

[206] CARLA Team. Carla simulator. https://carla.org/, 2022.

[207] Frank Tip. A survey of program slicing techniques. *J. Program. Lang.*, 1995.

[208] Suratose Tritilanunt. Performance evaluation of non-parallelizable client puzzles for defeating dos attacks in authentication protocols. In *Proc. DBSec*, 2010.

[209] Chia-che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. Civet: An efficient java partitioning framework for hardware enclaves. In *Proc. USENIX Security*, 2020.

[210] U.S. Department of Transportation (US DOT). Connected Vehicle Pilot Deployment Program. https://tinyurl.com/y29u9czy, 2019.

[211] U.S. Department of Transportation (US DOT). Intelligent Transportation Systems - Connected Vehicle Basics. https://tinyurl.com/yxjj98vr, 2019.

[212] U.S. Department of Transportation (US DOT). Intelligent Transportation Systems - Connected Vehicle Basics - DSRC. https://tinyurl.com/y5spr5cb, 2019.

[213] U.S. Department of Transportation (US DOT). Intelligent Transportation Systems - Connected Vehicle Pilot Deployment Program. https://tinyurl.com/yy5u7am6, 2019.

[214] U.S. Department of Transportation (US DOT). ITS Standards Program — Standards Group. https://tinyurl.com/yyzb8n4g, 2019.

[215] Veins. Vehicles in network simulation. https://veins.car2x.org/.

[216] Velodyne. Velodyne lidar. https://tinyurl.com/mc2duyeb, 2015.

[217] VENTOS. Vehicular network open simulator. http://maniam.github.io/VENTOS/, 2019.

[218] Matthew Wagner and Bruce McMillin. Cyber-physical transactions: A method for securing vanets with blockchains. In *IEEE PRDC*, 2018.

[219] Lev Walkin. ASN.1 Compiler. http://lionet.info/asn1c/, 2019.

[220] Ziwen Wan, Junjie Shen, Jalen Chuang, Xin Xia, Joshua Garcia, Jiaqi Ma, and Qi Alfred Chen. Too afraid to drive: Systematic discovery of semantic dos vulnerability in autonomous driving planning under physical-world attacks. In *Proc. NDSS*, 2022.

[221] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *Proc. IEEE S&P*, 2022.

[222] Tsun-Hsuan Wang, Sivabalan Manivasagam, Ming Liang, Bin Yang, Wenyuan Zeng, and Raquel Urtasun. V2VNet: Vehicle-to-vehicle communication for joint perception and prediction. In *Proc. ECCV*, 2020.

[223] Ziran Wang, Guoyuan Wu, and Matthew J. Barth. A review on cooperative adaptive cruise control (CACC) systems: Architectures, controls, and applications. In *Proc. ITSC*, 2018.

[224] Mark Weiser. Program slicing. In *Proc. ICSE*, 1981.

[225] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. Plug-n-pwned: Comprehensive vulnerability analysis of OBD-II dongles as A new over-the-air attack surface in automotive iot. In *Proc. USENIX Security*, 2020.

[226] William Whyte, Jonathan Petit, Virendra Kumar, John Moring, and Richard Roy. Threat and countermeasures analysis for WAVE service advertisement. In *Proc. IEEE ITSC*, 2015.

[227] Wai Wong, Shihong Huang, Yiheng Feng, Qi Alfred Chen, Z Morley Mao, and Henry X Liu. Trajectory-Based Hierarchical Defense Model to Detect Cyber-Attacks on Transportation Infrastructure. In *Transportation Research Board 2018 Annual Meeting (TRB)*, 2019.

[228] Tichakorn Wongpiromsarn and Richard M Murray. Formal verification of an autonomous vehicle system. In *Conference on Decision and Control*, 2008.

[229] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *Proc. IEEE/ACM ASE*, 2013.

[230] Qixue Xiao, Kang Li, Deyue Zhang, and Weilin Xu. Security risks in deep learning implementations. In *Proc. IEEE S&P Workshops*, 2018.

[231] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In *Proc. CHARME*, 1999.

[232] Daniel Zelle, Christoph Krauß, Hubert Strauß, and Karsten Schmidt. On using TLS to secure in-vehicle networks. In *Proc. ARES*, 2017.

[233] Kexiong (Curtis) Zeng, Shinan Liu, Yuanchao Shu, Dong Wang, Haoyu Li, Yanzhi Dou, Gang Wang, and Yaling Yang. All your GPS are belong to us: Towards stealthy manipulation of road navigation systems. In *Proc. USENIX Security*, 2018.

[234] Qingzhao Zhang, David Ke Hong, Ze Zhang, Qi Alfred Chen, Scott Mahlke, and Z Morley Mao. A systematic framework to identify violations of scenario-dependent driving rules in autonomous vehicle software. *Proc. ACM SIGMETRICS*, 2021.

[235] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. {PeX}: A permission check analysis framework for linux kernel. In *Proc. USENIX Security*, 2019.

[236] Xumiao Zhang, Anlan Zhang, Jiachen Sun, Xiao Zhu, Y Ethan Guo, Feng Qian, and Z Morley Mao. Emp: Edge-assisted multi-vehicle perception. In *Proc. MobiCom*, 2021.

[237] Ziyuan Zhong, Zhisheng Hu, Shengjian Guo, Xinyang Zhang, Zhenyu Zhong, and Baishakhi Ray. Detecting safety problems of multi-sensor fusion in autonomous driving. *CoRR*, abs/2109.06404, 2021.

[238] Ziyuan Zhong, Gail E. Kaiser, and Baishakhi Ray. Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles. *CoRR*, abs/2109.06126, 2021.

[239] Lidong Zhou and Zygmunt J Haas. Securing ad hoc networks. *IEEE network*, 1999.