CERIAS Tech Report 2023-4
Proactive Vulnerability Identification and Defense Construction -- the Case for CA
by Khaled Serag Alsharif
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

PROACTIVE VULNERABILITY IDENTIFICATION AND DEFENSE CONSTRUCTION – THE CASE FOR CAN

by

Khaled Serag Alsharif

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

August 2023

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Dongyan Xu, Co-Chair

Department of Computer Science

Dr. Z. Berkay Celik, Co-Chair

Department of Computer Science

Dr. Antonio Bianchi

Department of Computer Science

Dr. Jing (Dave) Tian

Department of Computer Science

Approved by:

Dr. Kihong Park

To that summer night by the bonfire in Alabama.

ACKNOWLEDGMENTS

In embarking on this challenging and rewarding journey of completing my dissertation, I have been fortunate to have the support and guidance of many incredible individuals who have shaped my path and contributed to my growth. Their unwavering belief in my abilities, words of wisdom, and presence in both professional and personal capacities have been instrumental in my success.

First and foremost, I am eternally grateful to my family, who has been the bedrock of my achievements and a constant source of love, encouragement, and guidance. My Dad, the first male role model of my life, taught me moderation, self-reliance, and self-respect. I can still hear him saying in Arabic: "Toughen up, for luxury never lasts." My Mom, with her unshakable faith and steadfast support, instilled in me the virtue of unyielding determination. Her words, "Always sharpen your axe" and "Be slow but sure," have guided me through thick and thin. Hazem's dedication to guitar practice has taught me the power of consistency and discipline. It was an embodiment of the idea of atomic habits, way before James Clear made his best seller. Leena, Mazen, and Mama Fatima's constant belief in me has been an inexhaustible source of inspiration. Finally, my cousin Ayman's continuous presence in my life despite the distance and time difference, was evidence that contrary to what people believe, being out of sight does not inevitably mean out of mind.

On the academic front, I am immensely grateful to my advisors, Professor Dongyan Xu, and Professor Z. Berkay Celik, for their invaluable guidance and mentorship. Professor Xu, your concise yet deep feedback on writing papers and constructing narratives has honed my skills and pushed me to excel. I will forever remember your words, "Always be self-critical. If you were reviewing your own paper, would you accept it?". Professor Celik, your unwavering support and belief in my abilities, even when I was down, have been transformative. I am grateful for the times when you went beyond the role of an advisor, lending an ear to my fears and providing guidance when I needed it the most. The short "Z." in your name, standing for Zeynel-Abedeen, carries a special significance to me as it is the same name as my beloved grandfather, who was always an inspiration in my life. I am thankful to Professor Dave Tian and Professor Antonio Bianchi for agreeing to serve on my final exam committee and for having served on my preliminary exam committee despite their busy schedules. Their valuable inputs have helped shape this dissertation.

I owe a debt of gratitude to my colleagues who have played a pivotal role in my academic journey. Rohit, thanks for giving me the seed for all my paper ideas. Vireshwar, thanks for being my first mentor and providing guidance during my formative years. I am grateful to all my lab mates, especially Sungwoo, Ozgur, Prashast, and Arslan for their constant support and willingness to lend a helping hand, even amid their busy schedules. Thanks for your selflessness and camaraderie.

Beyond the academic realm, I am thankful to my roommate, Ahmad Elkashif, for providing a sense of family away from my own family. I am also grateful to my Ph.D. friends, Ihem, Marwa, Nico, Armen, Brooke, Safa, Luisa, Yeni, Ata, and many others. Thank you all for being there in good times and bad, providing support, laughter, and a sense of belonging during this demanding journey. I would also like to express my appreciation to my lifelong friends. Amr, you have been the brother that my mother never gave birth to, and I am grateful for your enduring friendship. Makram, Tohamy, and Hesham, your presence in my life has brought joy and camaraderie. From my time in Saudi Arabia, I am grateful for having met Muath and Musaab in my life. Muath, you have been the most loyal friend I have ever had. Musaab, thank you for remaining my friend throughout the years, despite the ups and downs. Your friendship is truly cherished. To my best friends in Alabama, Danny, John, and Patrick, I am forever indebted for your generous support and for helping me make the decision to embark on this Ph.D. journey; had it not been for you, I would have never made it.

Lastly, I would like to express my indebtedness to my colleagues from my time at Boeing, Zach, Josh, Adam, and Wayne. You were instrumental in helping me realign my thoughts after a period of protracted wandering. I can trace back many of my current principles to our conversations on different topics. Jeff., thanks for being the best manager I have ever had. Your constant support throughout my employment and even after I left the company has been invaluable.

Funding Acknowledgement. I would like to thank the Office of Naval Research (ONR) for their financial support. This work was supported in part under Grants: N00014-22-1-2671 and N00014-18-1-2674. Any opinions, findings, and conclusions in this dissertation are those of the authors and do not necessarily reflect the views of the ONR.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The progressive integration of microcontrollers into various domains has transformed traditional mechanical systems into modern cyber-physical systems. However, the beginning of this transformation predated the era of hyper-interconnectedness that characterizes our contemporary world. As such, the principles and visions guiding the design choices of this transformation had not accounted for many of today's security challenges. Many designers had envisioned their systems to operate in an air-gapped-like fashion where few security threats loom. However, with the hyper-connectivity of today's world, many CPS find themselves in uncharted territory for which they are unprepared.

An example of this evolution is the Controller Area Network (CAN). CAN emerged during the transformation of many mechanical systems into cyber-physical systems as a pivotal communication standard, reducing vehicle wiring and enabling efficient data exchange. CAN's features, including noise resistance, decentralization, error handling, and fault confinement mechanisms, made it a widely adopted communication medium not only in transportation but also in diverse applications such as factories, elevators, medical equipment, avionic systems, and naval applications.

The increasing connectivity of modern vehicles through CD players, USB sticks, Bluetooth, and WiFi access has exposed CAN systems to unprecedented security challenges and highlighted the need to bolster their security posture. This dissertation addresses the urgent need to enhance the security of modern cyber-physical systems in the face of emerging threats by proposing a proactive vulnerability identification and defense construction approach and applying it to CAN as a lucid case study. By adopting this proactive approach, vulnerabilities can be systematically identified, and robust defense mechanisms can be constructed to safeguard the resilience of CAN systems.

We focus on developing vulnerability scanning techniques and innovative defense system designs tailored for CAN systems. By systematically identifying vulnerabilities before they are discovered and exploited by external actors, we minimize the risks associated with cyber-attacks, ensuring the longevity and reliability of CAN systems. Furthermore, the defense mechanisms proposed in this research overcome the limitations of existing solutions, providing holistic protection against CAN threats while considering its performance requirements and operational conditions.

It is important to emphasize that while this dissertation focuses on CAN, the techniques and rationale used here could be replicated to secure other cyber-physical systems. Specifically, due

to CAN's presence in many cyber-physical systems, it shares many performance and security challenges with those systems, which makes most of the techniques and approaches used here easily transferrable to them. By accentuating the importance of proactive security, this research endeavors to establish a foundational approach to cyber-physical systems security and resiliency. It recognizes the evolving nature of cyber-physical systems and the specific security challenges facing each system in today's hyper-connected world and hence focuses on a single case study.

1. INTRODUCTION

Over the course of the past few decades, an increasing number of once purely mechanical systems have come to be exceedingly computerized to the point where they cannot be conceptualized as mechanical systems anymore. These systems are currently known as cyber-physical systems. Cyber-Physical Systems (CPS) have information processing, sensing, control, and actuation components. They also have wired or wireless communication systems to exchange information and commands. The process these mechanical systems underwent to become cyber-physical systems was incremental, spontaneous, and often unpremeditated. Concretely, when the computerization process first started, designers did not foresee these systems as future "cyber-physical systems." Rather, they viewed them as the same mechanical systems they have always had but could improve by adding a digital sensor here, a processor there, or a digital connection in-between. Consequently, many of these gradually-added components lacked any security attributes since designers did not anticipate those systems to operate in the hyper-connected environment of today.

A perfect example of this progression is how the Controller Area Network (CAN) revolutionized the operation of many cyber-physical systems, especially for vehicles. In the 1970s, the evolution of integrated circuits eventually led to the development of microprocessors and microcontrollers. To have an edge on their competitors [1], certain vehicle manufacturers started making the first attempts at using microcontrollers on vehicular platforms in the form of electronic control units (ECUs). Initially, the number of ECUs in a vehicle was insignificant. Accordingly, vehicle manufacturers used point-to-point connections for inter-ECU communication. As time went by, more and more vehicle functions started becoming computerized. Gradually, wiring started posing a significant challenge stunting the increase in the number of ECUs and, by that very fact, the computerization of vehicle operation. Several attempts were made at devising a communication protocol for vehicle ECUs to reduce the amount of wiring. Between 1983 and 1986, Bosch developed the Controller Area network, or the CAN bus as an answer to the growing wiring question [2]. CAN further offered low-cost and efficient solutions to the most challenging questions of many cyber-physical systems, such as interference, priority management, decentralization, error handling, and fault confinement [3]. Inevitably, CAN started witnessing wide-scale acceptance and in 1991, Mercedes-Benz W140 was the first to introduce CAN into a production vehicle. Since its introduction, CAN has established

itself as the main internal communication medium for vehicles. Additionally, due to its success, it found its way into several other cyber-physical systems, from avionics and maritime applications to factories, elevators, and medical equipment. Thanks to CAN, many previously mechanical systems have now become cyber-physical systems.

Similar to many systems designed to improve mechanical systems, early CAN designers did not anticipate their network to operate in today's hyper-connected environment with all the security risks that it entails but rather in an almost air-gapped environment where security should not be a concern. Hence, although reliable and robust, CAN lacks any security measures. A compromised ECU has all that it needs to launch an array of attacks, including sending fake data and impersonating other ECUs. While this security deficiency may not have posed a grave threat in an age where the only way to access the CAN bus was via constant physical access, usually granted only to authorized users, it represents a severe menace to today's increasingly connected and computerized cyber-physical systems. A perfect example of how these factors could come into play and jeopardize modern cyber-physical systems is the famous Charlie Miller and Chris Valasek's 2015 Jeep hack. The two security researchers used the cellular network to remotely access the vehicle's CAN bus via a compromised ECU, demonstrating almost complete remote control abilities over the car below certain speed limits, including popping the locks, controlling the steering wheel, and disabling the brakes. While the incident was very costly for Jeep, which had to recall 1.4 million cars, it shed light on the importance of security for today's vehicles and cyber-physical systems in general.

Unfortunately, the security of cyber-physical systems today is still in a state of precarity. On the one hand, vulnerabilities are still found accidentally, usually by external actors. Those actors could be malicious and intend to cause actual harm, unlike security researchers such as Miller and Valasek whose goal was to bring the importance of automotive security to attention by shedding light on its vulnerabilities. Similarly, unlike security researchers, most attackers will not reveal that they have successfully breached a system since they want their breach to go undetected for as long as possible to maximize the benefits they could reap from it. On the other hand, security is still not prioritized except in response to manifest breaches in the sense that on several occasions, the process of determining the system's most suitable security measures does not begin until the exploit has already taken place or, sometimes, is even ongoing. Such a flawed approach to security has many faults. First, a security measure is not always found promptly. Thus, more similar breaches

could come about until it is found since many people now know the vulnerabilities of the system and how to exploit them. Second, even if a defense approach is promptly found, and in the case of widely deployed systems, recall costs are usually punitive, as we saw with the Jeep hack. Lastly, most of the defense approaches found under such tight timing requirements often come in the form of patches whose performance impact on the system's performance is often inadequately assessed and which fail to treat the root cause of the problem but instead its symptoms. Frequently, this leads to the same vulnerability later manifesting somewhere else, only to be patched again. This endless process of patching, which could continue indefinitely, may eventually render the entire system unusable due to the sheer number of patches.

In this dissertation, we propose a proactive vulnerability identification and defense construction approach to security. By proactive vulnerability identification, we mean that we should not wait for the vulnerabilities of the system to reveal themselves, but proactively and intentionally scan for them. Further, the process in which we look for vulnerabilities should be systematic. Even if, one day, a vulnerability accidentally reveals itself, we should systematically and pre-emptively investigate its root cause to look for other vulnerabilities that may stem from the same root cause later in the future. For the defense construction side, the aim should be to find defense approaches that consider the nature of the system, especially its performance, and that defend against several vulnerabilities at once, not a set of patches that tackle each vulnerability separately.

We take CAN as a case study for this approach. However, the same approach could be applied to other systems as well. We first show how we can identify new vulnerabilities in a seemingly old standard by building a vulnerability scanning tool that systematically looks for loopholes in unexplored areas of CAN. After identifying new vulnerabilities, we show how malicious actors could exploit these vulnerabilities by constructing a multi-stage attack that employs all the uncovered vulnerabilities. We evaluate it on a testbed and a real CAN bus of a test vehicle. Next, we show how to build a defense system that considers the nature of CAN and its performance challenges, such as the length of messages and their worst-case delays, while simultaneously protecting against the most prevalent CAN attacks, including the attacks that we discovered in the vulnerability identification phase, by capitalizing on a simple idea that utilizes an existing and unused channel for our security purposes. Finally, we evaluate our defense on a testbed and a real CAN bus and show that it achieves exceptional success rates in defending against a wide array of attacks.

We argue that CAN is a good case study for this approach for several reasons. First, CAN's presence in many cyber-physical systems, ranging from drones and vehicles to airplanes and navy ships and from medical equipment and elevators to factories and power stations makes it a good entry point to the vulnerabilities of many cyber-physical systems. It also makes the ideas and techniques used here easily transferable to other cyber-physical systems since they share similar performance and security challenges. Second, CAN's long-serving and wide-scale deployment gives people the illusion that there are no more vulnerabilities to discover in it. This dissertation shows that this is a wrong assumption. As such, every system that has not yet undergone a vulnerability assessment process should take heed and bring its security posture into question, no matter how long they have been operating. Lastly, CAN's wide-ranging use cases and performance requirements, thanks to its widespread adoption, complicate the process of designing a defense system for it. In other words, CAN is a non-trivial use case as there are many aspects that designers should consider before concocting security measures for it. In sum, designing a good defense system for CAN does not involve thinking about security only but many other performance requirements as well.

1.1 Contributions and Dissertation Outline

The outline of this dissertation is as follows. In Chapter 2, we provide the necessary background on CAN. In Chapter 3, we embark on our vulnerability identification mission: we explain the rationale and motivation behind our vulnerability scanning approach, we show how we used it to uncover new vulnerabilities, and finally, we explain how we used the discovered vulnerabilities to construct exploits and validate their feasibility by evaluating them on two platforms. Chapter 4 details our defense construction process: it begins by identifying the current problems of the existing defense approaches and explains why all of them have so far failed to achieve wide-scale adoption, it then proceeds to lay out its approach to securing CAN traffic, it follows that by an analysis of its impact on message delays, overhead, memory, as well as its expected security impact against several attacks, and finally, it evaluates several security and performance aspects on two platforms. In total, this Dissertation makes the following main contributions:

- We show how to systematically scan for new vulnerabilities in the CAN standard by following the upcoming steps. First, we identify a possibly vulnerable area of the CAN standard, namely

its error handling and fault confinement mechanism. Next, we build a dynamic vulnerability identification tool named CAN Operation Explorer (CANOX) for CAN's error handling and fault confinement mechanism, which studies the behavior of a test node in a controlled test environment. To the best of our knowledge, this is the first attempt at designing such a tool for CAN.

- We identify three major vulnerabilities in the CAN standard. Each of which could be used separately by the attacker to launch various attacks against various ECUs on the CAN bus. To show the feasibility of exploiting these vulnerabilities, we construct an end-to-end multistaged attack that utilizes all the uncovered vulnerabilities.

- We evaluate the multistage attack on a testbed and a real CAN bus of a test vehicle. We achieve very high attack success rates on both platforms.

- We identify the problems and gaps in the existing CAN defense systems and approaches which render the vast majority of these defenses ineffective at securing CAN systems.

- We build a defense system named Zero-Byte CAN (ZBCAN) that endeavors to address all the identified problems and gaps. Concretely, the defense system attempts to defend against the most common CAN attacks, including the attacks that we proposed in the vulnerability identification phase, and to offer attack detection as well as prevention abilities using a simple idea that leverages an existing CAN channel that has never been previously used for defense purposes. Simultaneously, the defense system tries to consider CAN's nature by avoiding using any fields of its already-short messages, abandoning the use of computationally heavy operations such as message authentication codes, analyzing its impact on the delays and schedulability of messages on the bus, and ensuring that its impact on those aspects remains within the acceptable bounds.

- We evaluate several performance and security aspects of our defense system on a testbed and a real vehicle's CAN bus. We achieve outstanding attack detection and prevention rates against a variety of attacks. We also validate the accuracy of our performance analysis and show that our system's impact on memory, overhead, message delays, and schedulability is acceptable even for devices with restricted memory and computational resources.

1.2 Publications

The chapters of this dissertation are drawn from, partially or fully, the following publications:

- **ZBCAN: A Zero-Byte CAN Defense System.** *Khaled Serag*, Rohit Bhatia, Akram Faqih, Muslum Ozgur Ozmen, Vireshwar Kumar, Z. Berkay Celik, Dongyan Xu. In Proceedings of the the 32^{nd} USENIX Security Symposium, 2023.

- **Attacks on CAN Error Handling Mechanism.** *Khaled Serag*, Vireshwar Kumar, Z. Berkay Celik, Rohit Bhatia, Mathias Payer, Dongyan Xu. In Proceedings of the NDSS' Fourth International Workshop on Automotive and Autonomous Vehicle Security (AutoSec), 2022

- **Exposing New Vulnerabilities of Error Handling Mechanism in CAN.** *Khaled Serag*, Rohit Bhatia, Vireshwar Kumar, Z. Berkay Celik, Dongyan Xu. In Proceedings of the 30^{th} USENIX Security Symposium, 2021

2. BACKGROUND

2.1 Normal CAN Operation

Architecture of a CAN ECU. As shown in Fig. 2.1, a CAN node or an Electronic Control Unit (ECU) consists of three major components: an application program, a CAN controller, and a CAN transceiver. The application program writes/reads message data and its identifier (ID) to (from) the controller. The controller is responsible for framing, bus arbitration, sending/receiving acknowledgments, and error handling. Lastly, the transceiver translates the bitstream coming out of the CAN controller into the differential voltage signal that is transmitted on the bus.

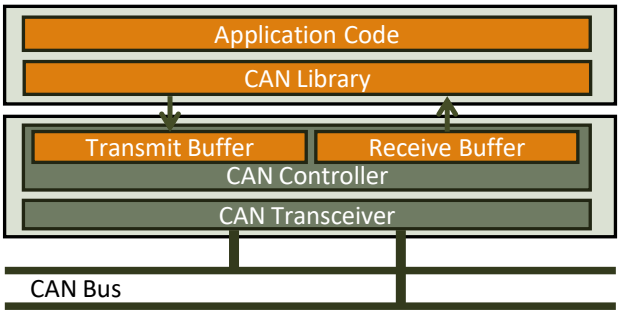

Figure 2.1. Architecture of an ECU.

CAN Basics. CAN is a broadcast-based bus that uses a *publish-subscribe* communication model. It uses differential voltage signaling to represent zeros (dominant) and ones (recessive). The transceiver communicates a bit $(0/1)$ on the bus using a two-level (high/low) voltage value. As such, the bits 0 and 1 are called dominant and recessive bits, respectively. During concurrent transmission of different bits by two or more nodes, the bus acts as a wired-AND gate, e.g., when a dominant bit and a recessive bit are concurrently transmitted, the resulting bit on the bus is dominant.

Framing. Two data frame formats could be used, the standard and the extended formats. As shown in Fig. 2.2, in the standard format, the ID is 11 bits long. The ID does not indicate the source/destination of the message, but it describes the meaning of the data contained in the message. Hence, a receiver ECU cannot determine the source. Although not intended to have any security impacts, this fact works as a double-edged sword, it facilitates impersonation attacks, but at the same time provides anonymity to the transmitter.

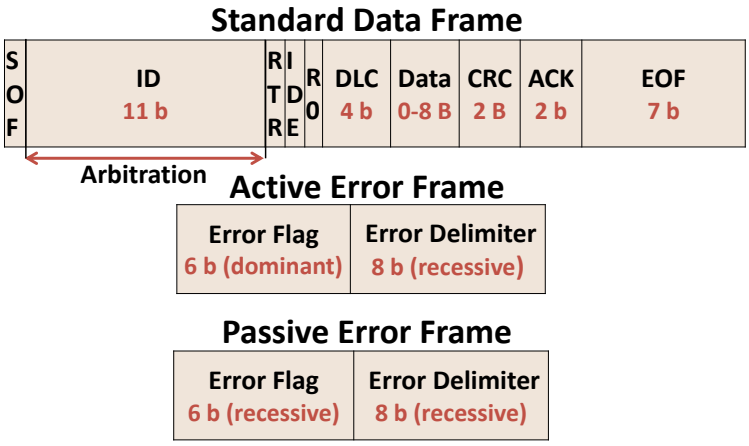

Figure 2.2. Different CAN frames.

Arbitration and Priority. At the beginning of every CAN message, there is an ID field. Often, an ECU has multiple message IDs, but an ID has only one transmitter. CAN uses lossless bitwise arbitration to resolve collisions and provide transmission priorities. If two nodes start transmitting at the same time, they first go through an arbitration phase, starting at the ID field and ending at the RTR bit, as shown in Fig. 2.2. CAN controllers sense the bus as they transmit every bit. During the arbitration phase, if a controller sending a recessive bit senses that the bus is dominant, it stops the transmission. Consequently, this mechanism gives messages with a smaller ID value a higher priority and guarantees that beyond arbitration, there is, at most, one transmitter. After arbitration, if a controller sending a recessive bit senses that the bus is dominant, it stops the transmission and raises an error. Errors and their impact are further explained later in this section.

Figure 2.3. CAN frame fields of two back-to-back frames.

Inter-Frame Spacing (IFS). Messages conclude by sending 7 ones called the End Of Frame (EOF) sequence (Fig. 2.3). Following EOF, any ECU that wishes to transmit a new frame will have to wait for 3 additional bits called the *Inter-Frame Spacing (IFS)*.

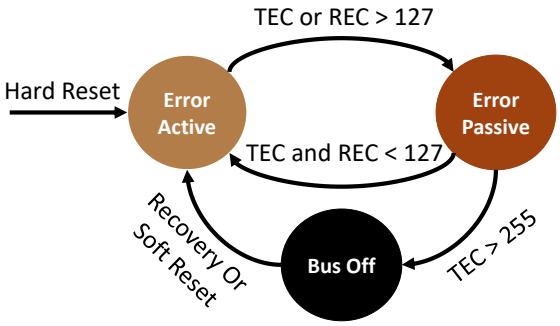

Figure 2.4. Error states in CAN.

2.2 Error Handling and Fault Confinement

CAN Errors. CAN defines five error types: Bit Errors, Stuff Errors, Form Errors, Acknowledgement Errors, and CRC Errors. These errors may happen either during transmission or reception. Each node maintains two counters: Transmit Error Counter (TEC) and Receive Error Counter (REC). When a transmitter encounters an error, it sends an error frame and increases TEC by 8. Similarly, when a receiver encounters an error, it sends an error frame and increases REC by 1. A successful transmission decreases TEC by 1, and a successful reception decreases REC by 1. The format of error frames differ based on the error state of the node.

Error States. To provide fault confinement, CAN defines three error states. Each error state is associated with a set of behavioral rules regarding message transmission, reception, and the way they handle errors. The transition between error states is based mainly on the internal error counters of the CAN controller as illustrated in Fig. 2.4.

(1) The Error Active State: A node is in this state by default. Here, a node's minimum idle time between two consecutive frames is 3 bit-periods. Additionally, in this state, when the node witnesses an error, it sends an *active error frame*, consisting of 6 dominant bits, followed by 8 recessive bits (Fig. 2.2). *Active error frames* override and terminate any ongoing transmission.

(2) The Error Passive State: A node enters this state when its REC or TEC exceeds 127. Here, an additional 8-bit suspend transmission period is added between successive transmissions. This period technically lowers the priority of successive and retransmitted messages to prevent a faulty node from consuming the busload in scenarios such as the *babbling idiot*. Further, on witnessing an error, the node transmits a *passive error frame*, consisting of 14 recessive bits (Fig. 2.2). Unlike the *active error frame*, a *passive error frame* is not observable on the bus and does not interrupt any ongoing transmission. This prevents faulty nodes from interrupting the transmission of healthy ones.

(3) *The Bus-Off State:* A node enters this state when its TEC exceeds 255. In this state, the node disconnects itself from the network. It stops transmitting or receiving messages. The node is permitted to go back to the *active error* state after observing at least 128 instances of 11 recessive bits on the bus. On re-entering the error active state, all error counters reset to 0.

2.3 Threat Model

Before delving into the main body of this dissertation, we first need to state the attacker abilities considered throughout the dissertation. We assume that the attacker can control all of the OSI layers of an ECU on the bus, except the physical and data-link layers of the CAN standard. This means that the attacker could execute arbitrary code on the ECU but does not control its CAN protocol controller or transceiver. The attacker here can inject or read *full messages* into or from the CAN bus only through an uncorrupted CAN controller. As such, they would still abide by all the CAN physical and data-link rules. They cannot inject malformed messages or error frames. Neither can they read message bits directly from the CAN bus. Instead, they could only read the assembled message that is forwarded to them by the CAN controller after it has received it.

This has been the most widely used threat model in the CAN security literature [4–10] due to its practicality as it does not require specialized hardware or physical access. Both requirements have traditionally been considered too strong. The attacker could achieve such abilities by remotely gaining access to an existing ECU on the bus using channels such as Wi-Fi, Cellular, Bluetooth, and radio [4–7, 11] or by exploiting diagnostic software or hardware infrastructures [10, 12–14].

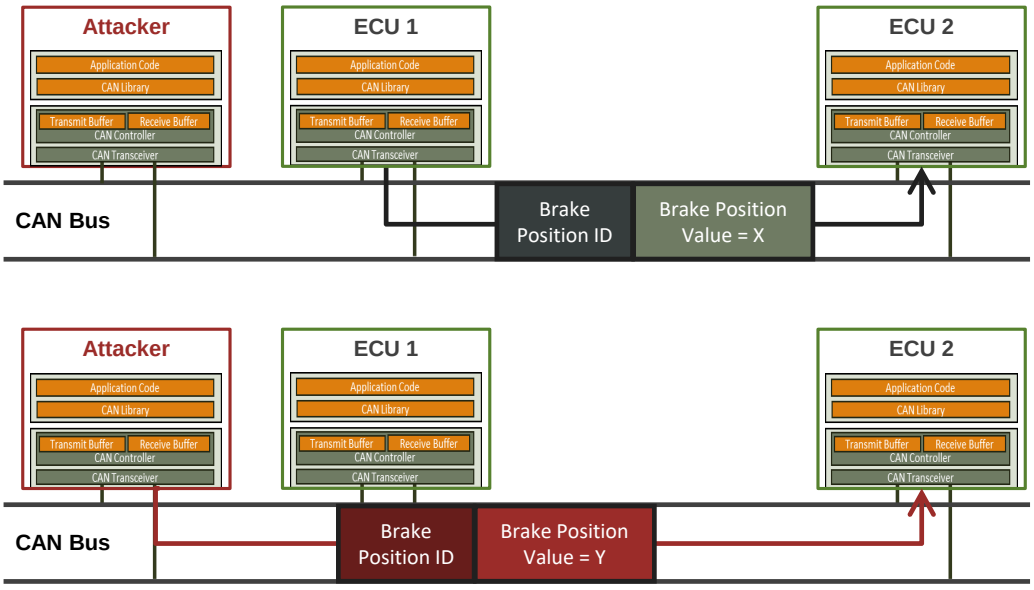

Figure 2.5. Illustration of a targeted injection (impersonation) attack in which the attacker observes how a legitimate ECU transmits its messages and then imitates it.

Although recently, the prevalence of OBDII ports and the design of many modern vehicles have made physical access relatively easier [13, 15, 16], and although some of the latest research works have shown [17, 18] that several ECU types could be manipulated by an attacker to achieve partial or full control over the datalink layer, this remains an implementation problem pertinent to the automotive or ECU manufacturer and not a problem with the specification of the CAN standard itself whose analysis is the focus of this dissertation.

2.4 Common CAN Attacks

Injection Attacks. Attacks where a malicious ECU injects messages into the bus. We broadly consider the following as injection attacks:

(1) Targeted Injection: The attacker here forges and injects messages that abide by the formatting rules of *existing messages* that are in charge of certain functions in order to alter those specific functions. Impersonation attacks, where the attacker tries to mimic the identity of the legitimate ECU are a sub-type of targeted injection (Fig 2.5).

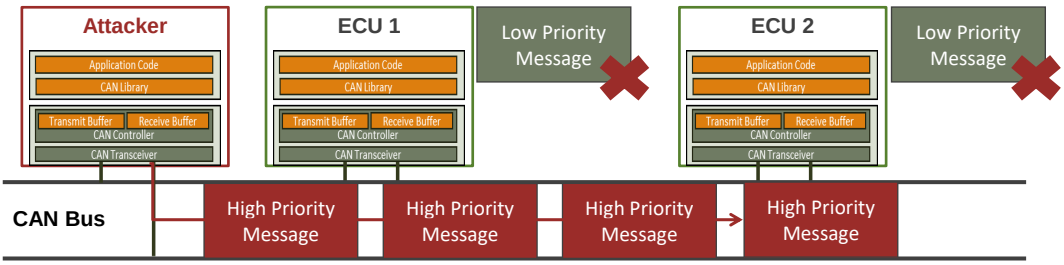

Figure 2.6. Illustration of a flooding attack.

(2) Replay Attacks: The attacker Replays one or more messages transmitted by a different ECU.

(3) Random Injection: The attacker here forges IDs randomly or semi-randomly to cause damage or to discover hidden message semantics (e.g. *fuzzing attacks*).

Flooding Attacks. As shown in Fig. 2.6, the attacker's goal here is to cause an availability problem. One way to do that is to inject an endless stream of back-to-back high-priority messages. The impact of that is that other ECUs with lower priority messages will be unable to access the bus and eventually drop their messages. Flooding could be targeted or absolute. Targeted flooding is when the attacker floods the bus with messages of a specific priority that is higher than that of certain messages on the bus, but not others. For instance, an attacker may flood the bus with messages with an $ID = 0x20$, thus depriving all messages with $ID > 0x20$ access to the bus but allowing messages with smaller IDs. On the other hand, absolute flooding is when the attacker denies all messages access to the bus. This could be done by sending messages with a higher priority (smaller ID) than any other message on the bus. The simplest way to do that is by sending messages with an $ID = 0x0$. This causes all other ECUs to be unable to send, and eventually drop, their messages.

Error Handling Attacks. An ever-widening attack surface, these attacks cause a victim ECU to encounter errors and increase their error counters in order for the attacker to achieve various purposes. For example, by accumulating these errors, attackers could push ECUs to the *error passive* or *bus-off* states. These error states could then be exploited to launch further attacks such as

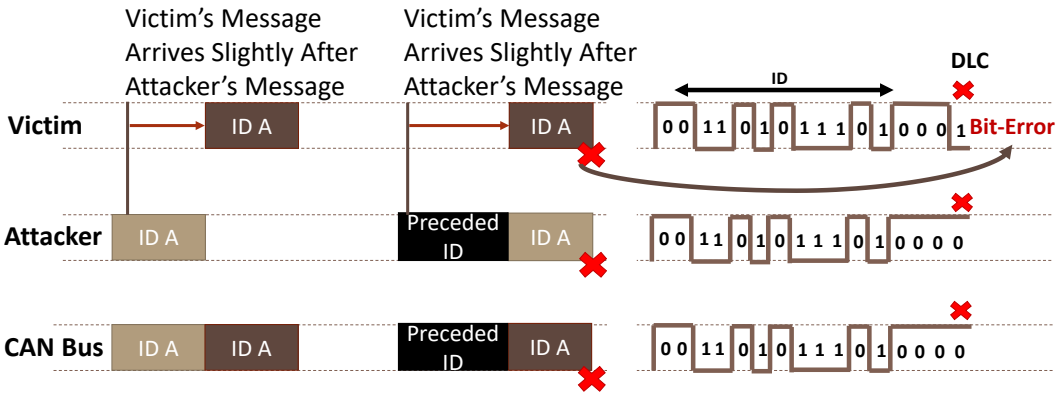

Figure 2.7. Enabling time synchronization between the attacker's and the victim's messages by using a preceded ID message to facilitate packet collisions on the CAN bus.

persistent DoS attacks or to map the network, as will be shown in this dissertation. They could also be used to evade voltage intrusion detection systems [19] as proposed in other works.

Simultaneous Transmission. A technique used by attackers to inject collisions and increase the error counter of a victim. Since the CAN standard uses ID arbitration to prevent two nodes from transmitting simultaneously, the attacker here needs to transmit a message with the same ID as a target message exactly at the same time but with a different payload. This causes both the attacker and the victim to think that they won the arbitration, only to experience an error later in the message and increase their TEC by 8. For an attacker to target and induce a collision with a victim's message, the attacker needs to *simultaneously* transmit a message with the same ID as the victim's message, but with a different payload. Hence, the attacker first needs to estimate the arrival time of the victim's message, and then attempt to transmit precisely at the expected arrival time.

For a periodic message, attackers could expect the arrival time of a message by monitoring it and calculating its period. However, messages on the bus encounter small jitter in transmission time, which may cause the attacker's message to arrive slightly earlier or later than the victim's message. To address this challenge, in [20], the authors propose employing a *preceded ID message*, that has a higher-priority ID than the victim's message and is transmitted (by the attacker) immediately before the transmission of the victim's messages. As shown in Fig. 2.7, this enforces both the

victim and the attacker to start transmitting exactly at the conclusion of the preceded ID message, synchronizing the victim's and the attacker's messages.

3. EXPOSING NEW VULNERABILITIES OF ERROR HANDLING MECHANISM IN CAN

3.1 Motivation

A vehicle today contains over a hundred ECUs sensing and actuating most vehicles' maneuvers. However, prior research has shown that an attacker can gain access to a vehicle's CAN by compromising an in-vehicle ECU (e.g., telematics control unit) through a wired/wireless medium, including USB, cellular, Bluetooth and WiFi [4–7, 13]. Since CAN was not designed with security in mind, a compromised ECU can be exploited to launch various attacks on other safety-critical ECUs (e.g., brakes), which cannot be directly compromised [8]. In this project, we study an alarming type of attacks that directly exploits the *error handling* and *fault confinement* mechanism of CAN, turning CAN's reliability function into its security weakness [20–23].

On a vehicular CAN, collisions, interference, and wire faults occur often. To operate for extended periods with no external supervision, CAN defines a set of rules for error detection, handling, and fault confinement to be enforced by every node on the bus *throughout its operation* [3]. For fault confinement, a CAN node monitors its health by counting the number of encountered errors. Additionally, CAN introduces the concept of *error states*, which are different sets of rules governing transmission, reception, and error signaling. CAN defines three error states: *error active*, *error passive*, and *bus off*. By default, nodes operate in the *error active* state. Once a node's error counter exceeds a certain threshold, it enters the *error passive* state, where stricter rules are enforced. If errors persist, the node transitions itself into the *bus off* state, where it disconnects itself from the network. Exploiting this specific feature, prior work presented a denial-of-service (DoS) attack called *bus off attack* [20] in which an attacker node deliberately collides its packets with those of a victim node, causing *bit errors*. These errors gradually increase the victim's error counter until it drops into the *bus off* state, disconnecting it from the bus.

This dissertation claims that the attacker's ability to induce packet collisions in CAN is extremely dangerous as it opens the doors for attackers to *dictate the victim's error state*. We argue that, since CAN nodes were *not* expected to leave the *error active* state except under certain error conditions, the security impacts of operating outside of the *error active* state are vastly understudied, and the vulnerabilities inherent to their design remain undiscovered.

In this project, we introduce CANOX (CAN Operation eXplorer), an automated testing tool that explores the impacts of operating outside of the default *error active* state to identify possible vulnerabilities in the Controller Area Network (CAN) standard. CANOX places a CAN node in a controlled environment, sets its operation and error state, systematically changes the operational conditions of the node and the environment, and monitors certain behavioral metrics to identify conditions that result in unexpected node behaviors.

Using CANOX, we have discovered three fundamental vulnerabilities in CAN's error-handling mechanism. (1) *Passive Error Regeneration:* The error signaling procedure in the *error passive* state could make the node's error counter rapidly and silently increase under normal bus conditions. An attacker could exploit this vulnerability to launch an advanced DoS attack that we call the *Single Frame Bus Off* (SFBO), in which the attacker pushes a node to the *bus off* state by attacking a *single message*, making it more than 36x faster than previous *bus off* attacks. (2) *Deterministic Recovery*: When a node recovers from the *bus off* state, it exhibits a deterministic behavior. An attacker could exploit this vulnerability to prevent a node's recovery from the *bus off* state, perpetuating the node's stay in a state of disconnection. (3) *Error State Outspokenness*: A node operating in the *error passive* state exhibits a distinct, easily identifiable behavior. An attacker could exploit this vulnerability to identify message sources or identify an ECU's function.

Even though an attacker may exploit each of these vulnerabilities individually, we demonstrate the significant threat of these vulnerabilities by combining them to construct a single, powerful, multi-staged attack called the *Scan-Then-Strike attack* (STS). In STS, a remote attacker, with no knowledge of the car's internals, exploits the discovered vulnerabilities to gain knowledge before striking their victim. First, the attacker starts by mapping the internal network. Next, the attacker identifies a safety-critical ECU. The attacker then learns the ECU's recovery behavior. Finally, the attacker strikes the ECU and prevents it from recovering, achieving a persistent DoS. In contrast to the *Original Bus Off Attack* (OBA) [20], STS utilizes SFBO to push a victim to the *bus off* state by attacking a *single message*, enabling it to be persistent, as it can immediately re-attack the victim's recovery attempts. Moreover, OBA assumes that the attacker already knows the network map, ECU functions, and the IDs they transmit. In comparison, STS exploits the discovered vulnerabilities to gain this knowledge, significantly reducing the attacker's assumptions.

Prior efforts have proposed different network mapping solutions [24–29]. Nevertheless, these works approached network mapping from a defense standpoint. Thus, they either required physical access and special equipment [26–29], or used time-consuming learning techniques that worked only with periodic messages [24, 25], and proved to be evadable [19, 30]. Conversely, to the best of our knowledge, STS employs the first network mapping solution that identifies sources of periodic and aperiodic messages with 100% accuracy without using special equipment but using the existing ECU capabilities. We summarize our contributions as follows:

- Developing CANOX, an automated testing tool to examine CAN's error handling and fault confinement mechanism to find vulnerabilities in the CAN standard.
- Discovering three major vulnerabilities in CAN's error handling and fault confinement mechanism that could be exploited separately or in combination. We combine them and construct a powerful and persistent attack (STS) that showcases how all these vulnerabilities could unfold together.
- Demonstrating the practical impact and the platform-agnostic nature of the vulnerabilities by evaluating STS first on a testbed and then later on a real vehicle.

3.2 CANOX

CAN Operation eXplorer (CANOX) is an automated testing tool, which explores the impacts of operating outside of the default *error active* state to detect possible vulnerabilities in the CAN standard. The purpose of building CANOX is to assess what an attacker can achieve by pushing a node outside of the default *error active* state. Therefore, CANOX's main goal is to detect *unexpected* behavioral deviations from the *error active* state. To do so, CANOX places a fully controllable and programmable CAN node, called the Node Under Test (NUT), in different error states, sets up specific test scenarios, and defines its expected behavior in each scenario. It then monitors the node's behavior and flags any deviations from its expected behavior.

While changing the error state of a node affects certain behavioral aspects specified in the CAN standard [3], the implications of such changes and the enforcement level of error state-specific rules have not been thoroughly analyzed. Therefore, we use CANOX to investigate unexpected behaviors that result from conflicting and unenforced rules, or hidden implications of poorly studied rules, as these may pose significant threats to the security and performance of a CAN system. Here we

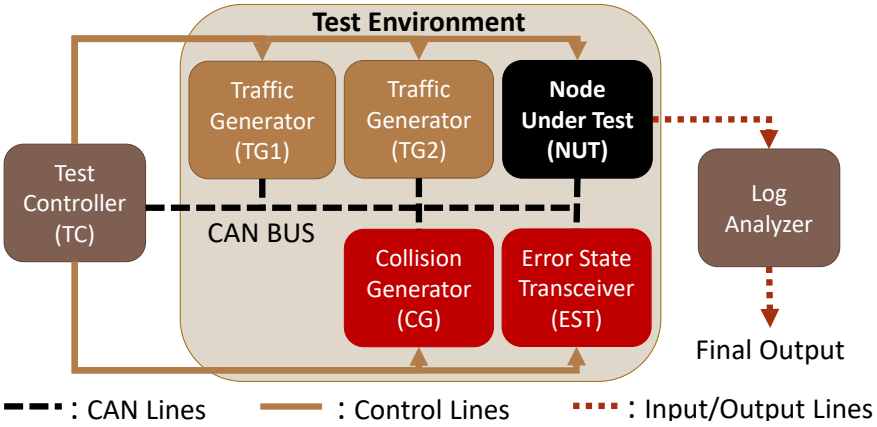

Figure 3.1. Architecture of CANOX.

define "unexpected" as a behavioral deviation from the *error active* behavior that exceeds a specific *threshold*. We quantify the behavioral metrics in Sec. 3.2.1 and explain the thresholds required to find the deviations in Sec. 3.2.2. Using CANOX, we uncover three new fundamental vulnerabilities in CAN's error handling and fault confinement mechanism discussed in Sec. 3.3.

3.2.1 Architecture and Operation

CANOX consists of a *Test Controller* (TC) connected to a controlled test environment containing a *Node Under Test* (NUT), as shown in Fig. 3.1. The test environment includes a CAN bus connected to three different components. The first component is the *Collision Generator* (CG), which generates packet collisions per the request of TC by injecting a message with the same ID as the NUT's message, as explained in Sec. 2. The second component is the *Error State Transceiver* (EST), used by TC to directly read bits from the bus and to set the NUT's error state by directly injecting error frames. The last two components include two *Traffic Generators* (TG), used to generate the CAN traffic given the message priority and the bus load. TC is connected to the CAN bus and to all components of the test environment to control their operations. NUT is placed inside the test environment and controlled by TC. NUT logs its measured performance metrics into a file and sends it to the *Log Analyzer*. The log analyzer analyzes the NUT's logs and flags the conditions leading to anomalous changes in NUT's behavior compared to its behavior at default *error active* state.

Behavioral Metrics. CAN error states directly impact two behavioral aspects: *error signaling*, and *transmission delay penalties* in certain scenarios. Hence, we expect a change in the error state would result in a change in these two aspects: (1) error frames and (2) transmission delays. We need two metrics to quantify these behavioral changes. One challenge is to monitor *passive error frames* since they are composed solely of recessive bits indistinguishable from the idle bus; hence they are unobservable. To overcome this challenge, we monitor the Transmit Error Counter (TEC) because it reliably indicates the presence of errors, even if they are unobservable on the bus. Hence, we define two evaluation metrics: (1) *Standby Delay* (SD), to monitor transmission delays, defined as the delay between the moment the message is buffered and marked as ready for transmission, and the moment it is successfully transmitted. (2) *TEC Value Change* (TECC), to monitor error frames, defined as the change in the TEC value before and after the message is transmitted.

Test Scenarios. CAN specifies different sets of rules governing message transmission and error signaling in different error states. While most of these rules are similar, they differ in specific cases. Specifically, CAN imposes certain delay penalties on *passive* nodes sending successive messages or retransmitting failed messages. Moreover, CAN dictates that *passive* nodes signal errors using *passive error frames* as opposed to *active error frames* when they witness errors. Therefore, we set up three test scenarios covering these cases to exhaustively assess the behavioral differences between an *error passive* node and an *error active* node under different bus conditions:

(1) Single Transmission: We set NUT to send a single message periodically, and record the SD and TECC for every transmission. This enables us to assess the impact of additional penalties on message transmissions in *passive* nodes.

(2) Single Collision: We set the NUT to experience errors during its message transmissions, causing its transmissions to fail and forcing it to retransmit the failed messages. NUT periodically sends a single message. However, the collision generator induces a *single collision* every time NUT sends a new message. Single means that the collision generator causes a collision to NUT's initial transmission attempt, but does not cause any further collisions to its retransmissions. Finally, for every message transmission (including all of its retransmission attempts), NUT logs the SD and

Algorithm 1 Test Controller Algorithm

```
1: L ← {0%, ..., 100%}
2: P ← {lower, higher, mixed}
3: S ← {active, passive}
4: points = L × P × S
5: rounds ← {1, ..., nrounds}
6: scenarios ← {single, collision, successive}
7: SDA ← Empty SD Array of Arrays
8: TA ← Empty TECC Array of Arrays
9: for s in scenarios do
10:     for p in points do
11:         for r in rounds do
12:             Turn off CG, TG, and NUT
13:             Set state of NUT
14:             Adjust TG to load and priority
15:             if (scen = collision) then
16:                 Turn CG on
17:             Start NUT operation
18:             for ntrans in transmissions do
19:                 Record SD and TECC
20:             Compute Average SD for the round
21:             Compute Average TECC for the round
22:         SDA_{s,p} ← Average SD across rounds
23:         TA_{s,p} ← Average TECC across rounds
24:     Pass SDA_s to Analyzer
25:     Pass TA_s to Analyzer
```

TECC. This scenario enables us to monitor the impact of the altered error signaling mechanism and assess the impact of the delay penalties imposed against failed message retransmissions.

(3) Successive Transmission: We set the NUT to periodically send two back-to-back messages to assess the impact of the additional delay penalties imposed against back-to-back transmissions in *passive* nodes. We mark the second message as ready for transmission immediately after the first message is transmitted successfully. Here, NUT records the SD and TECC for the *second message* in every transmission cycle.

CANOX Operation. For each scenario, the test controller sets NUT's error state using the error state transceiver. It also sets the traffic load and its priority using the traffic generators. It then enables NUT to start transmitting. We describe this process in Algorithm 1. We repeat each scenario

Algorithm 2 $\texttt{Analyzer}(\texttt{SDArray}, \texttt{TECCArray}, \texttt{Th}_{\texttt{SD}}, \texttt{Th}_{\texttt{TECC}})$

```
loads ← {0%, ..., 100%}
priorities ← {lower, higher, mixed}
SDA ← SDArray
TA ← TECCArray
for p in priorities do
    for l in loads do
        if (SDA_{p,l,passive} > (SDA_{p,l,active} + Th_SD)) then
            Flag SDA_{p,l} for both states
        if (TA_{p,l,passive} > (TA_{p,l,active} + Th_TECC)) then
            Flag TA_{p,l} for both states
if (SDArray has any flagged elements) then
    Plot all average SD readings for the scenario
if (TECCArray has any flagged elements) then
    Plot all average TECC readings for the scenario
```

for a number of rounds (nrounds) for every error state (S), and every traffic load (L) and priority (P). Each round, NUT sends ntrans pairs of messages and logs the SD and TECC for each transmission. After each scenario is terminated, the log analyzer reads the logs and compares SD and TECC for each priority and bus load pair in the *passive* case to the *active* case as described in Algorithm 2. The log analyzer flags the scenario and plots the result if any *passive* metrics differ from the *active* metrics more than the specified thresholds. Threshold selection is detailed in Sec. 3.2.2.

Equipment. Our Node Under Test (NUT) comprises an Arduino Uno board connected to a CAN bus shield. The CAN bus shield contains an MCP2515 CAN controller and an MCP2551 CAN transceiver. We use one test controller (TC), one collision generator (CG), two traffic generators (TG), and one error state transceiver (EST). TC, CG, and each of the two TGs comprise an Arduino Uno board connected to a CAN bus shield. We use an MCP2551 as the error state transceiver. To generate deliberate packet collisions, we use the method described in Sec. 2.2. We achieve "mixed priority" by having one traffic generator send high-priority traffic while having the other generator send low-priority traffic. For communication between the test controller and different test components, we use the CAN bus and boards' digital pins.

3.2.2 Test Parameters

Test Input Generation. The input space for testing scenarios becomes intractable with two states, three scenarios, more than 2^{29} traffic priority levels, and an unlimited number of bus loads. To reduce complexity, we restrict the priority levels and bus loads.

First, we select only three points for priority levels: High, Low, and Mixed. These priority levels are justified by the fact that relative to NUT, any external message on the bus could be categorized as having either a higher or lower priority than NUT's messages. Note that higher priority means traffic with a lower ID value than NUT's messages, and lower priority means traffic with a higher ID value. However, we add an intermediate point of mixed traffic since the traffic usually is not strictly higher or lower in normal bus operations.

Second, to reduce the input space of bus loads, we select five loads: $0\%, 25\%, 50\%, 75\%$, and 100%. These busloads are justified because, from NUT's perspective as it attempts to transmit, it views the bus as either idle or busy. Nonetheless, the bus is never always full (100%) or empty (0%) in normal bus operations. Thus, we add three additional intermediate points between bus empty and full to comprehensively observe behavioral trends. Overall, we reduce the input space into five bus loads, three priority levels, and two states to be tested in scenarios without losing generality.

Behavioral Metric Threshold Selection. The expected node behavior is different from a scenario to another. Therefore, we configure TECC and SD thresholds to different values for different scenarios. We use the CAN standard's specifications [3] to specify the metric thresholds for each scenario.

In the *single transmission* scenario, the *active* node starts with an initial $TEC = 0$, while the *passive* node starts with an initial $TEC = 159$. The standard does not define a time penalty on single message transmissions against *active* or *passive* nodes. Therefore, we expect the standby delay difference threshold between states Th_{SD} to be $0\mu s$. For TEC, the standard states that each successful transmission reduces the TEC counter by 1 if TEC is $0 < TEC < 256$. Since only the *passive* node's TEC lies within that range, we expect this rule to apply only to the *passive* node. Hence we set the TECC difference threshold between states Th_{TECC} to 1.

In the *single collision* scenario, the standard defines a penalty of 8 bit-periods ($16\mu s$ at $500kbps$) against *passive* nodes' retransmissions. *Active* nodes, on the other hand, do not have this penalty

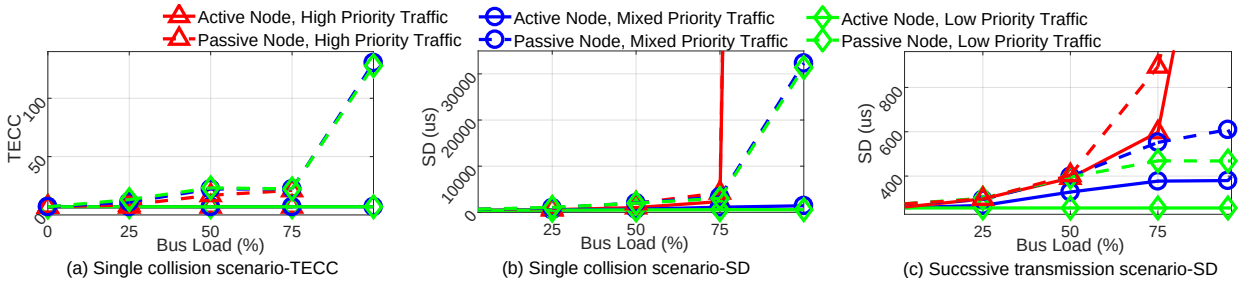

Figure 3.2. TEC change and standby delay values for the scenarios identified by CANOX as having an unexpected behavior.

imposed against them. Hence, we expect the standby delay threshold between the two states Th_{SD} to be $16\mu s$. For TEC, the standard states that each collision increases TEC in both states by 8, and that each successful transmission reduces it by 1. Since these two rules hold true for both states, we expect the TECC threshold between the two states Th_{TECC} to be 0.

In the *successive transmission* scenario, the *active* node starts at $TEC = 0$. The $passive$ node starts at $TEC = 159$. The standard defines a penalty of 8 bit-periods ($16\mu s$ at $500kbps$) against *passive* nodes' back-to-back transmissions, while no penalties are imposed against *active* nodes. Hence we set the standby delay threshold between states Th_{SD} to $16\mu s$. For TEC, the standard states that each successful transmission reduces TEC by 1, for $0 < TEC < 256$. Since only the *passive* node's TEC lies within that range, we set the TECC threshold between states Th_{TECC} to 1.

Calibration. Depending on the equipment and the time measurement method used, delay calculation may be slightly inaccurate. To account for such inaccuracy, the maximum possible deviation from the actual value should be calculated and added to the standby delay threshold Th_{SD}. Initially, in our experiments, the maximum observed deviation was $7.5\mu s$. However, later in our experiments, we optimized the code corresponding to time measurement. This reduced this error margin to $< 3\mu s$. Similarly, when specific CAN controllers experience a collision while $248 < TEC < 256$, they set TEC to 0 instead of increasing it by 8, as they do not allow the TEC value to go above 256. This may result in the *passive* node having a slight deviation in its average TECC from the expected value. Depending on the sample size, the maximum deviation resulting from this case should be

calculated and added to the TECC threshold (Th_{TECC}). Initially, in our experiments, the maximum observed deviation was ≈ 0.03. However, later in our experiments, we filtered out samples with an initial $TEC \geq 248$. This reduced the error margin to 0.

3.3 Discovered Vulnerabilities

CANOX detected that both the *single collision* and *successive transmission* scenarios yield unexpected behaviors. In the *single collision* scenario, CANOX detected that the average SD and TECC difference between *error active* and *error passive* states violated the specified thresholds under multiple testing conditions, which we analyze in this section. For the *successive transmission* scenario, CANOX also detected multiple violations of the SD threshold. However, it did not detect any violations of the TECC threshold. For the *single transmission* scenario, CANOX did not detect any unexpected behavior as the TECC and SD values remained below the specified thresholds for all bus load and priority pairs. Fig. 3.2 illustrates the discrepancies for the *single collision* scenario's TECC and SD, and the *successive transmission* scenario's SD. Below, we further analyze the plots and provide details of each discovered vulnerability.

3.3.1 Passive Error Regeneration

Detection. In the *single collision* scenario, CANOX detected that the *passive* node violated the given TEC change threshold Th_{TECC} for all priorities and bus loads $\geq 25\%$. As shown in Fig. 3.2a, we observe that the *active* node had a fixed TECC value (i.e., 7) regardless of the bus load or priority. Whereas the TECC value for the *passive* node was dependant on the bus load but not the priority. Further, we observe that the *passive* node had a TECC of 128 at a 100% bus load. *This means that at* 100% *bus load, the node went from the error passive to the bus off state after encountering a single collision.* We explain the reason behind this peculiar behavior below.

Test Results Explanation. Among the above observations, we highlight two findings. (1) Certain silent (*passive*) errors were present on the bus, visible only to the *passive* node. (2) These errors pushed the node from the *error passive* to the *bus off* state. After consulting the standard, the dynamics of these findings can be explained as follows.

A passive error frame consists of 14 *recessive bits* as shown in Fig. 2.2. However, the number of recessive bits at the end of a frame is 8, and the minimum bus idle time is 3 bit-periods [3]. This implies that the minimum number of recessive bits between the dominant acknowledgment bit of one frame and the dominant start-of-frame bit of any other frame on the bus is *only* 11 *bit-periods*, which is *shorter* than the time needed to transmit a passive error frame. Now, in the *single collision* scenario, when the *passive* node encounters a collision, it tries to transmit a passive error frame after the dominant acknowledgment bit of the frame involved in the collision. However, as the voltage levels for the recessive bit of the passive error frame and the idle bus are the same, other nodes on the bus fail to detect that the *passive* node is transmitting a passive error frame. Because the bus is busy, other nodes start transmitting messages before the conclusion of the passive error frame. This causes an error in the delimiter part of the passive error frame interrupting its transmission. This interruption is interpreted by the *passive* node as a *form error*, resulting in the node raising its TEC by 8 and attempting to signal the *new* error by sending a *new passive error frame*. However, the new error frame is also interrupted in the same manner as the first frame. This continuous cycle repeats until the node's TEC reaches 256 pushing the node into the *bus off* state.

Vulnerability Description. The CAN standard states that for a *passive* node to terminate its *passive error frame* correctly, the bus must be idle for at least an additional 3 bit-periods between two consecutive frames. However, *the standard fails to provide a way of enforcement or explain the consequences of not fulfilling this rule* [3, 31]. CANOX reveals that due to the discrepancy between an error frame length, and the minimum number of recessive bits required between two consecutive frames, *this rule cannot be enforced*. The consequences of this failure lead to what we call the *passive error regeneration vulnerability*. Exploiting this vulnerability, an attacker can interrupt a victim's passive error frame by transmitting a seemingly benign message frame. As such, this vulnerability allows it to *silently turn one error into a series of errors.*

Exploit 1: Single Frame Bus Off Attack (SFBO)

Exploit. We exploit the passive error regeneration vulnerability to craft a novel DoS attack called the Single Frame Bus Off (SFBO). Using SFBO, an attacker targets only one frame from the victim

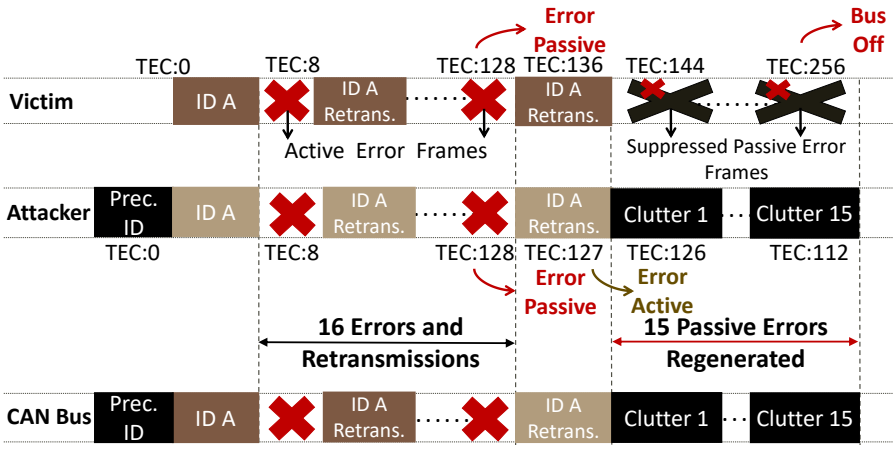

Figure 3.3. Illustration of the single frame bus off attack exploiting the passive error regeneration vulnerability.

to successfully push it to the *bus off* state where it cannot transmit or receive any messages. SFBO proceeds through four steps as described in Fig. 3.3.

Step-1: The attacker first targets a victim's message with a known ID, and forges a message with the same ID as the victim's ID but with a *higher priority content*. Throughout the dissertation, a *higher priority content* means a content of a shorter length or more leading zeros. Conversely, a *lower priority content* is either longer or with fewer leading zeros.

Step-2: The attacker then transmits the forged message simultaneously with the target message causing a deliberate collision, as explained in Sec. 2. Since the victim's message content has lower priority, it encounters a bit-error. This forces the victim to stop transmission of its message, and transmit an active error frame. Then, due to the victim's active error frame consisting of dominant bits, the attacker encounters a bit-error. The attacker stops message transmission and joins the victim in transmitting an active error frame. Moreover, according to the CAN standard, the automatic retransmission feature of a node is enabled by default. This means that the CAN controllers of both the victim and attacker retransmit their failed messages. Unfortunately, this leads to 16 back-to-back collisions. After each such collision, both of them increase their TEC values by 8. Hence, after the 16^{th} collision, both fall into the *error passive* state with a TEC value of 128.

Step-3: The message retransmission attempts by the victim and attacker continue for one more round. However, in this round, the victim generates a passive error frame that does not interrupt

the attacker's message. This allows the attacker to transmit their message successfully. Hence, the attacker decreases its TEC by 1 and gets back to the *error active* state.

Step-4: This is the point where the attacker exploits the passive error regeneration vulnerability. At this step, the attacker causes an error in the victim's *passive error frame* that was generated in the previous step by sending a message with an arbitrary ID. We refer to such a message as a *clutter message*. As such, the attacker sends 15 back-to-back clutter messages. This causes a regeneration of passive error frames, and the victim's TEC increases rapidly by 8 after every message until it reaches 256. This way, the attacker succeeds in pushing the victim to the *bus off* state by targeting a single message in a single attack round. We note that, if external higher-priority messages get transmitted while the attack is taking place, the attack will not be interrupted but instead *helped*, as the higher-priority traffic will play the same role as the clutter messages. In this case, the attacker may carry out the attack with fewer clutter-messages.

Impact. In the existing DoS attack (OBA) [20], the attacker follows the first three steps described for SFBO, to push the victim to the *error passive* state. Thereafter, the attacker needs to induce collisions in rounds of attacks; each round takes an entire periodic transmission cycle, with at least 18 new victim's messages (rounds) to push the victim from the *error passive* state to the *bus off* state. *On the contrary, the attacker in* SFBO *immediately exploits the passive error regeneration vulnerability* to push the victim to the *bus off state* in the same attack round. *This reduces the number of attack rounds from a minimum of 19 to a maximum of 1.* Hence, the impacts of SFBO are profound, not only because of its speed in pushing the victim to the *bus off* state, but also because this swiftness allows the attacker to keep the victim in the *bus off* state persistently, as discussed in Sec. 3.3.2. In Sec. 3.5.3, we provide a comprehensive comparison between SFBO and OBA.

Exploit 2: Setting Victim's TEC

Exploit. The passive error regeneration vulnerability can be used to *easily* set the TEC of a victim node to a chosen value between 135 and 256. This can be done by following the first three steps of SFBO, but controlling the number of clutter messages (denoted by $\mathtt{N_{Clutter}}$) in Step 4, as

discussed in Sec. 3.3.1. When $\texttt{N}_{\texttt{Clutter}} = 15$, the victim falls into the *bus off* state. However, for any $\texttt{N}_{\texttt{Clutter}} < 15$, the victim's TEC can be calculated as $\texttt{TEC}_{\texttt{Victim}} = 135 + (8 * \texttt{N}_{\texttt{Clutter}})$.

Impact. The ability to selectively set the victim's TEC value provides the attacker with nearly full and immediate control of the victim's error states. The applications of such an exploit are versatile. For example, in Sec. 3.3.3, we explain how this exploit plays a critical role in identifying a message source and extending it to map the entire network.

3.3.2 Deterministic Recovery Behavior

Detection. In the *single collision* scenario, CANOX detected that the *passive* node violated the given standby delay threshold Th_{SD} for all priorities with bus loads $\geq 25\%$. In Fig. 3.2b, we make three main observations. (1) The delay in the *passive* node is correlated with the bus load. (2) The SD curves in Fig. 3.2b are correlated with the TECC curves (Fig. 3.2a). (3) Most importantly, for low and mixed priorities, the *passive* node has an $SD \approx 31.7ms$ at 100% bus load.

Test Results Explanation. The CAN standard states that when a node goes to the *bus off* state, it stays there until observing at least 128 instances of 11 recessive bits on the bus. We validate that the SD value of $31.7ms$, mentioned in the third observation, is approximately equal to the time needed to observe 128 instances of 11 recessive bits. This points to a very interesting behavior revealed by CANOX. *After the node fails to transmit its message due to collision and enters the bus off state, the unsent message remains stuck in its CAN controller's transmission buffer and gets transmitted exactly when the node gets back to the error active state*.

Vulnerability Description. The CAN standard does not clearly define what to do with an unsent message if a node enters the *bus off* state. CANOX reveals that the node transmits such an unsent message at the exact moment it recovers (i.e., transitions back into the *error active* state). This allows an attacker launching a bus off attack to predetermine the ID and content of the messages sent by the victim at recovery and, as a consequence, how to attack the victim at recovery.

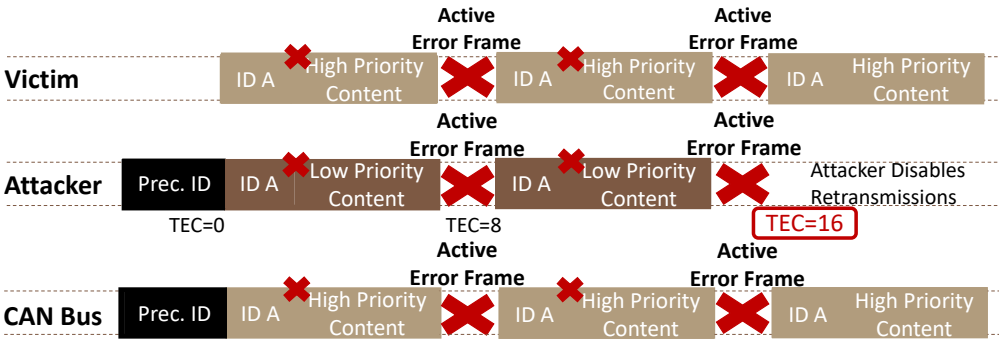

Figure 3.4. Behavior in the error active state.

Exploit: Persistent Bus Off

Exploit. An attacker may exploit the deterministic recovery vulnerability as follows. The attacker targets a victim's message ID, induces errors through collisions, and pushes the victim to the *bus off* state. This prevents the victim's message from being sent and pre-determines the ID and content of the message sent at the moment the victim recovers. Equipped with such information, the attacker can re-attack the message to prevent the victim's recovery, persistently pushing it into the *bus off* state. Hence, the attacker may persistently stop valid transmissions from the victim by first launching one instance of SFBO against a message, and then continuously launching instances of SFBO against every recovery attempt of the victim. We discuss how the attacker may estimate the victim's recovery time in Sec. 3.4.3.

Impact. The existing DoS attack, OBA, requires a long time to push the victim to the *bus off* state, and provides no clear way to prevent victim's recovery, rendering the DoS attack highly ephemeral. The deterministic recovery (coupled with the passive error regeneration) vulnerability poses a critical threat, as an attacker may exploit this to persistently prevent the node's recovery attempts as illustrated further in Sec. 3.4.

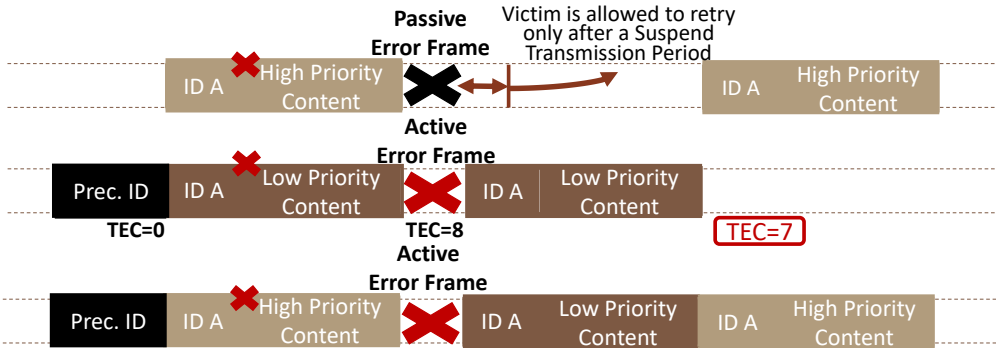

Figure 3.5. Behavior in the error passive state.

3.3.3 Error State Outspokenness

Detection. In the *successive transmission* scenario, CANOX detected that the *error passive* node violated the given standby delay threshold Th_{SD} for all low and mixed priority bus loads above 25%. In Fig. 3.2c, we make three main observations. (1) The difference in the SD values between *passive* and *active* nodes far exceeded the threshold for low and mixed priority bus loads above 25%. (2) The *passive* node had an extra SD of $\approx 240\mu s$ over the SD of the *active* node for low priority at 100% bus load. This delay is equivalent to one 8-byte message. (3) For low priority traffic, the SD of the *active* node was independent of the bus load.

Test Results Explanation. CAN imposes a *suspend transmission* penalty of 8 bit-periods on *passive* nodes in the cases of *successive transmissions and retransmissions*. This causes the *second message* in the *passive* node sending two successive messages to witness a *priority reduction*. This reduction causes the *second message* to *lose arbitration to any pending message* on the bus, even if the pending message has a lower priority ID. Hence, at high bus loads in the *successive transmission* scenario, the second message has an extra delay of around one message even in the case of low-priority traffic. In other words, lower than any message transmitted by a message in the *error active* state.

Vulnerability Description. CANOX reveals that a *passive* node will suffer from a priority reduction affecting successive message transmissions and retransmissions. *The priority reduction can be*

easily spotted and used by an attacker to differentiate between a message sent by an active node and a message sent by a passive node. We refer to this as the error state outspokenness vulnerability.

Exploit: Message Source Identification

The message source identification refers to the procedure for determining if two messages originate from the same victim. This can be achieved by first pushing the victim into the *error passive* state by using one message, and then determining if the source of the second message is in the *error passive* state. The victim can be pushed into the *error passive* state by exploiting the passive error regeneration vulnerability as discussed in Sec. 3.3.1. Below, we propose a novel technique to determine the error state of the victim over the bus by exploiting the error state outspokenness.

Determining Victim's Error State. An attacker can exploit this vulnerability to determine the victim's error state through the following four steps.

Step-1: The attacker forges a message with the same ID as the victim's message. However, as opposed to other techniques, the attacker here employs a *lower priority content*.

Step-2: The attacker induces a deliberate collision of their message with the victim's message. As such, the attacker encounters a bit-error since it transmits a recessive bit while the victim is sending a dominant bit. The attacker raises an *active error frame*, interrupting the victim's transmission. This causes both nodes to retransmit their messages.

Step-3: As illustrated in Fig. 3.5, if the victim is in the *error passive* state, it will not attempt to retransmit at the same time as the attacker due to the *suspend transmission period* penalty placed on its retransmissions. Hence, no further collisions will take place, and the attacker's message is successfully transmitted. This is followed by the victim's message. Conversely, as illustrated in Fig. 3.4, if another collision happens, it means that the victim is in the *error active* state. In this case, the attacker disables retransmissions to prevent further collisions.

Step-4: As a result of the previous step, if the attacker's TEC changes by only 7, they determine the victim to be in the *error passive* state. Otherwise, if the attacker's TEC changes by 16, the victim is considered to be in the *error active* state.

Impact and Applications. The applications of the source identification technique are manifold. For example, an attacker may use it to identify all the messages transmitted by a target ECU, identify an ECU's function, or map the entire CAN bus. All the aforementioned goals could help an attacker that wants to launch a targeted DoS attack or reverse engineer the network traffic to perform message injections. In Sec. 3.4.1, we explain how this exploit could be used to map an entire network. We note that this source identification technique is not limited to periodic messages, and could be used to map any message as long as its ID and arrival time are deterministic. Command-response messages and event-triggered messages are two examples of aperiodic messages that satisfy these conditions. We take advantage of this fact in the victim identification stage of the STS as discussed in Sec.3.4.2. To the best of our knowledge, *this is the first network mapping technique to map aperiodic messages without using special hardware.*

3.4 STS: Scan-Then-Strike Attack

To illustrate the impact of the discovered vulnerabilities, we develop an advanced multi-staged attack, Scan-Then-Strike Attack (STS), which exploits the combination of all discovered vulnerabilities. A remote attacker with no previous knowledge of the vehicle's internal network, number of ECUs, ECU functions, message formats, or IDs is able to: (1) map the internal network, determining the number of transmitting ECUs, and identify the sources of all periodic messages, (2) identify, among the mapped ECUs, an ECU that performs a safety-critical function, (3) learn how the ECU recovers from a DoS attack in the form of SFBO, and (4) launch a persistent DoS attack against the ECU by constantly relaunching continuous instances of SFBO against its recovery attempts.

STS differs from previous attacks in three aspects. First, it does not assume that the attacker is already knowledgeable of the vehicle's network map and safety-critical ECUs but rather gains this knowledge by exploiting the newly discovered vulnerabilities. Second, the immediate and swift nature of SFBO allows it to be launched against any ECU as opposed to the previous attacks that worked only against certain ECUs, as we will explain in Sec. 3.5.3. Lastly, its impact is persistent, as opposed to the previous volatile attacks.

3.4.1 Stage 1: Network Mapping

The first stage of STS is to perform the network mapping that relates the CAN bus messages to the transmitting ECUs. To do this, STS exploits the *error state outspokenness* vulnerability as explained in Sec. 3.3.3. Essentially, it performs checks on message pairs to see if they originate from the same ECU. This check is conducted by pushing the sender of one of the two messages to the *error passive* state, then checking the other message to see if it comes from an *error passive* ECU. We highlight that to successfully complete the check, it is critical to ensure that the sender stays in the *error passive* state until the completion of the check. However, satisfying this condition for a real-world ECU that sends multiple messages at different frequencies is challenging.

Consider an ECU that transmits two messages with different IDs, where one has a much longer period than the other. In this case, if the attacker pushes the ECU to the *error passive* state using the *short-period* message, the ECU would have transmitted many instances of this short-period message before any instance of the *long-period* message is transmitted. As a result, the successful transmission of the short-period messages brings down the TEC of the ECU, taking it back to the *error active* state before the check is completed. This invalidates the checking procedure. To address this challenge, the attacker should always pick the long-period message to push the ECU to the *error passive* state and then pick the short-period message to perform the check.

As shown in Algorithm 3, the network mapping stage of STS consists of the following steps. (1) The attacker records the bus traffic and makes a list of all the message IDs on the bus, sorted by their periodicity. (2) The attacker selects the message with the shortest period in the unassigned list of messages and assumes that a new ECU transmits it. (3) They select the message with the longest period in the unassigned list of messages and push its sender to the *error passive* state. (4) They check whether the selected shortest-period message is transmitted by an *error passive* ECU. If true, the selected longest-period message is assigned to the ECU sending the selected shortest-period message. If false, it is marked as *not* transmitted by the ECU. If the check is inconclusive, it is repeated. In all cases, the attacker waits for the TEC of the ECU (that they pushed to the *error passive* state) to go back to 0 since they do not want to push it to the *bus off* state unintentionally. The attacker repeats Steps 3-4 until the ECU is mapped to all its messages. Further, they repeat Steps 2-4 until all ECUs are fully mapped to their messages.

Algorithm 3 Network Mapping Algorithm

```
1: list ← Get list of ids and periods
2: Based on period, sort list
3: while list has unassigned ids do
4:     Get shortest − period unassigned id_small
5:     Create a new ecu_i, assign id_small to ecu_i
6:     while list has unchecked ids do
7:         Get longest − period unchecked id_big
8:         idBigResolved ← false
9:         while idBigResolved = false do
10:            Push id_big to Passive
11:            Check id_small state
12:            if id_small is passive then
13:                Assign id_big to ecu_i
14:                idBigResolved ← true
15:            else if id_small is active then
16:                Leave id_big unassigned
17:                Mark id_big as checked for ecu_i
18:                idBigResolved ← true
19:            else
20:                idBigResolved ← false
21:            Wait for TEC of the source of id_big to be zero
```

3.4.2 Stage 2: Victim Identification

In the network mapping stage, the attacker maps every ID to a specific sender. However, the attacker does not know the function of each sender. Here, the attacker's goal is to identify, among the mapped ECUs, the victim ECU that performs a specific safety-critical function (e.g., braking). To achieve that, STS exploits the *error state outspokenness vulnerability*, in addition to vehicle diagnostic protocols. Diagnostic protocols such as On-Board Diagnostics (OBD-II) define sets of request messages that trigger a response message from an ECU that performs a specific function. For example, a diagnostic message requesting information about the anti-lock braking system (ABS) will trigger a response from the electronic brake control module (EBCM).

Victim identification proceeds through the following four steps. (1) The attacker identifies a request to which the victim responds. For example, the VIN information comes from the ECM, transmission information comes from the TCM, and ABS information comes from the EBCM. The request message identification task could be carried out by acquiring an off-the-shelf OBD-II scanner, selecting the vehicle's make and model, selecting a specific vehicle function (i.e., ABS),

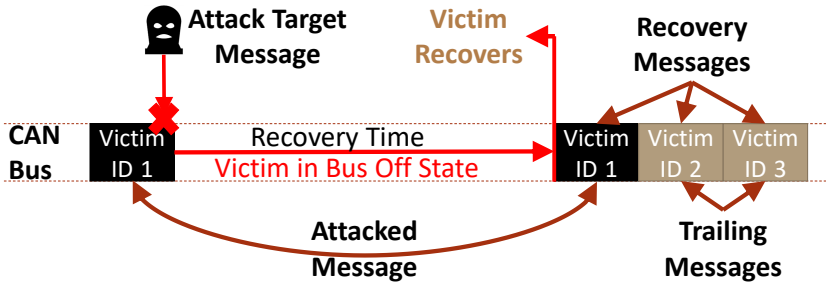

Figure 3.6. Illustration of the victim's recovery behavior.

and recording the request message sent by the scanner. This step could be carried out offline since its only goal is to identify the request message to which the target ECU responds. (2) They then send a forged request message on the CAN bus and measures the response time. (3) Next, they send another request message and, following the technique described in Sec. 3.3.3, they attack the response message, pushing its sender to the *error passive* state. (4) Finally, they check every mapped ECU to see which one is in the *error passive* state using one of its periodic IDs. This concludes the victim identification stage. Now that the attacker knows the victim ECU, it can be targeted using one of its periodic messages in the next stage of STS.

3.4.3 Stage 3: Learning Victim's Recovery

In this stage, STS exploits the *deterministic recovery vulnerability* to learn how the victim recovers. This enables the attacker to prevent the victim's recovery attempts and paves the path for a persistent DoS attack. To do that, STS needs to identify the victim's *recovery time* and the *recovery message* to be able to attack these attempts accordingly.

Recovery Messages. As discussed in Sec. 3.3.2, when an attacker pushes an ECU to the *bus off* state by attacking a message, the same attacked message will be transmitted at recovery. However, it does not always get transmitted alone. In many ECUs, especially those that apply long recovery intervals, additional messages will be buffered during the recovery interval. As a result, once the ECU recovers and sends the attacked message, it attempts to transmit all the other buffered messages. We call such buffered messages the *trailing messages*, which are shown in Fig. 3.6. Consequently, upon recovery,

the ECU transmits the attacked message followed by a number of trailing messages. STS exploits this fact to determine an *optimum ID* that can easily be attacked persistently in every recovery cycle. As such, the optimum ID needs to satisfy two conditions: (1) When attacked, it is the first recovery message. (2) When attacked, the first trailing message has the same ID. Usually, this condition will be satisfied if the attacker picks the ID with the shortest period (highest transmission rate) since it usually has the highest priority and probability of buffering within the interval. However, if an ECU has multiple IDs with the same period, the attacker must find which one satisfies these conditions.

Time Recovery Model. After an ECU enters the *bus off* state, it spends a specific time interval before getting back to the *error active* state. We call this interval the *recovery interval*. The CAN standard states that a bare minimum recovery interval corresponds to the time in which the ECU observes 128 instances of 11 recessive bits. However, many designers choose recovery intervals that are longer than that. As such, multiple recovery models exist on different ECUs. We identify the following four broad models, which can be specifically determined by launching multiple continuous instances of SFBO and observing the victim's recovery time.

1. *Bare Minimum:* The ECU recovers after observing 128 instances of 11 recessive bits, CAN's minimum requirement.
2. *Fixed:* The ECU recovers after a fixed recovery interval.
3. *Sequenced:* The recovery interval follows a sequence of different intervals. For example, the first time it goes into the *bus off* state, it recovers after $x\ ms$. If recovery fails, it reattempts recovering after $y\ ms$ such that $y \geq x$, and so on.
4. *Random:* The ECU recovers after a random interval. With no way to expect when the ECU (following this model) recovers, the attacker cannot suppress its recovery synchronously. Hence, we use the re-appearance of the attacked message to signal the ECU's recovery and attack the first trailing message. If the attacked message is the *optimum ID*, the first trailing message will have the same ID as the attacked message. This facilitates the attack, as the attacker does not need to guess and change the ID used in SFBO to match the trailing message at every recovery prevention instance.

Determining Victim's Recovery Model. The attacker can identify the victim's time recovery model by launching SFBO against the victim and observing the interval between the time it enters the *bus off* state and the time it recovers back to the *error active* state. To identify whether the recovery model is fixed, sequenced, or random, the attacker needs to launch another SFBO, wait until the victim attempts to recover, suppress its first recovery attempt, then let it recover again. It then measures the time spent by the victim in its second recovery attempt and uses it for comparison to determine the time recovery model as described in Fig. 3.7.

(1) Bare Minimum: If the time corresponds to 128 instances of 11 recessive bits, then this means that it follows a bare minimum recovery model.

(2) Fixed Interval: If the recovery time is constant in all instances, the model is determined to be the fixed interval. The attacker learns this interval by observing the time after attacking the victim once, letting it recover, and taking note of the amount of time it took to recover.

(3) Sequenced Intervals: In a sequenced intervals model, the victim uses a different interval every time the recovery is suppressed. However, the sequence of intervals is fixed. Here, the attacker can learn the sequence as it launches the attack. The attacker first uses SFBO against a victim message, then measures the first recovery interval in the sequence by letting the victim recover. Next, the attacker attacks the victim again and, using the learned interval, suppresses the first recovery, then learns the second interval in the sequence by letting the victim recover, and so on.

(4) Random: If there is no observable pattern in the victim's recovery time, the attacker considers the model to be random, even if the pattern is not truly random.

3.4.4 Stage 4: Recovery Prevention

Equipped with the information from the previous three stages, STS proceeds with this last, but the most critical stage where STS exploits SFBO to persistently prevent the victim's recovery. As opposed to previous DoS attacks that provided no way of achieving a persistent suppression of the

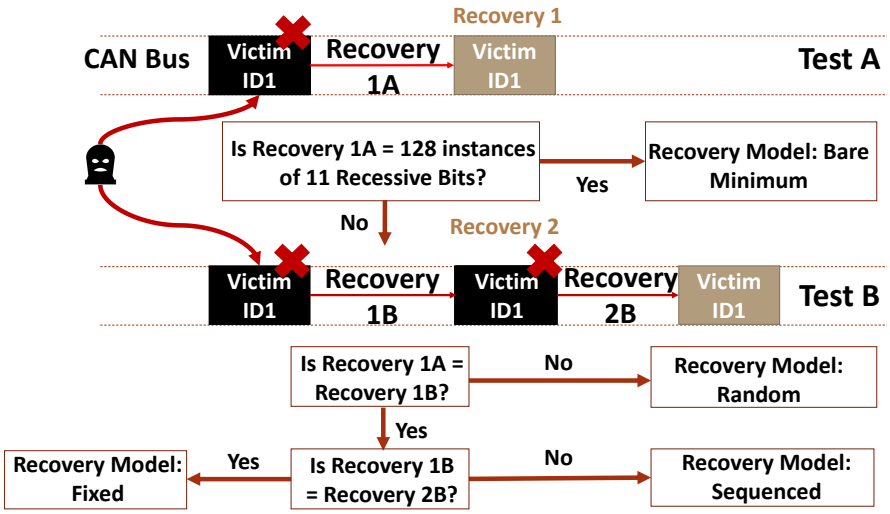

Figure 3.7. Determining victim's time recovery model

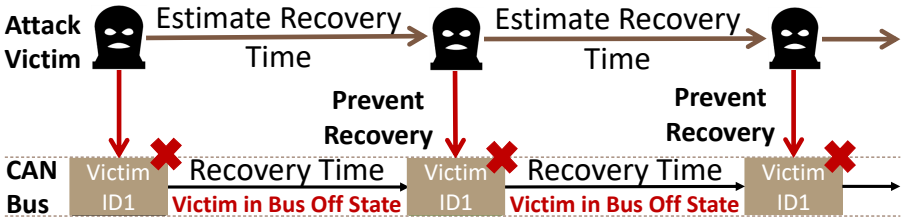

Figure 3.8. Demonstration of STS persistently preventing the victim's recovery from the bus off state.

victim, the swift nature of SFBO allows STS to realize such a goal. This stage proceeds through the following three steps. (1) The attacker launches an instance of SFBO against the victim's optimal ID (determined in Stage 3 of STS) as discussed in Sec. 3.3.2. (2) They predict the recovery time of the victim based on the time recovery model learned in Stage 3 of STS. (3) They prevent the victim's recovery by re-launching another instance of SFBO against the optimal ID. (4) They continuously loop around Steps 2 and 3 to suppress the victim persistently, as illustrated in Fig. 3.8.

Recovery Prevention for Different Models. Since there exist multiple *time recovery models*, the recovery estimation and prevention method differs between models. We now explain how to estimate and prevent recovery for each model.

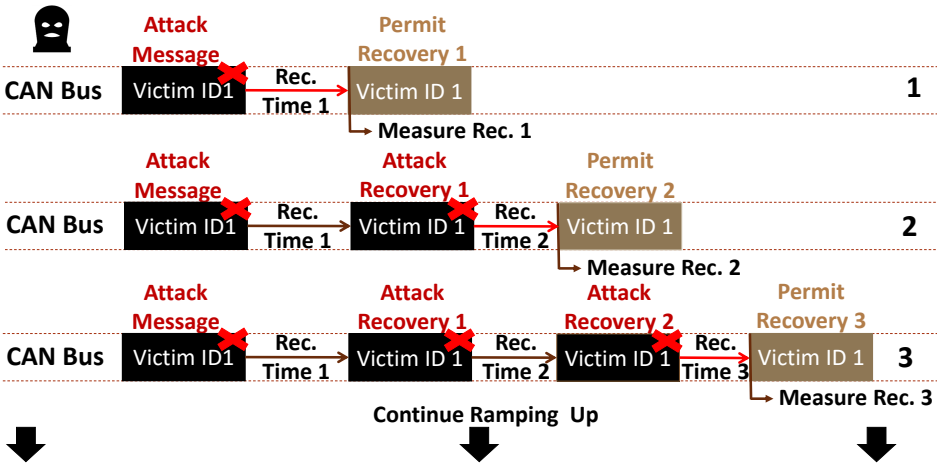

Figure 3.9. Ramping up suppression rate by learning more recovery sequences every iteration

(1) *Bare Minimum:* Recovery is estimated by observing 11 recessive bit instances since the last SFBO, then re-launching another SFBO instance by the 128^{th} instance.

(2) *Fixed Interval:* Recovery is prevented by measuring the time since the last SFBO instance and relaunching new instances after the determined fixed interval.

(3) *Sequenced Intervals:* The attacker prevents the recovery using the determined sequence of intervals, leading to a ramp-up attack that lasts longer at every recovery as shown in Fig. 3.9.

(4) *Random:* There is no way to expect when the ECU with this model recovers. Hence, the attacker cannot suppress the victim's recovery by attacking the first recovery message. Instead, we use the first message to signal the ECU's recovery and attack the first trailing message for recovery prevention as shown in Fig. 3.10. When a node recovers, it sends *recovery messages*, including the *attacked message*, and other *trailing messages*. By identifying the *optimum message ID* when identifying the victim's recovery behavior, the first trailing message will have the same ID as the attacked message. This facilitates the attacker's job, as it does not require guessing and changing the ID used in SFBO to match the trailing message at every recovery prevention instance.

Attacking trailing messages entails that the victim will be successful at transmitting the first recovery message at every recovery attempt. However, if the attacker chooses the ID with the shortest period to attack the ECU, the recovery time for the ECU will usually be much longer than the attacked messages period $\approx 10X$. This incurs a severe delay and transmission frequency

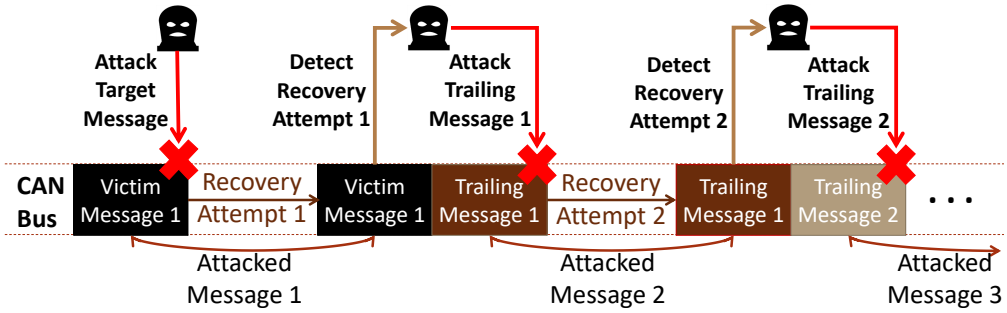

Figure 3.10. Suppressing victims with random recovery times by attacking trailing recovery messages.

Table 3.1. Network mapping results for ExpVehicle.

ECU #	IDs	ECU Function
ECU-1	0C5, 0C1, 1E5, 1C7, 1CD, 1E9, 184, 334, 2F9, 348, 34A, 17D, 17F, 773, 500	Electronic Brake Control Module (EBCM)
ECU-2	0F1, 1E1, 1F3, 1F1, 134, 12A, 3C9, 3F1, 4E1, 771, 4E9, 138, 514, 52A, 120	Body Control Module (BCM)
ECU-3	199, 0F9, 19D, 1F5, 4C9, 77F	Transmission Control Module (TCM)
ECU-4	0C9, 191, 1C3, 1A1, 2C3, 3C1, 3E9, 3D1, 3FB, 3F9, 4D1, 4C1, 4F1, 772	Engine Control Module (ECM)

reduction for the message ID under attack. We note that the data content of the attacked message will be stale because of the delay. Also, since an ECU usually transmits multiple message IDs, the attack will successfully block all of the non-attacked message IDs transmitted by the ECU.

3.5 STS Evaluation

Below, we report our results corresponding to each attack stage of STS evaluated on a CAN bus testbed and a real vehicle. Additionally, we compare the proposed attack, SFBO, with the Original Bus Off Attack (OBA) [20] in terms of swiftness, feasibility, and persistence.

3.5.1 Evaluation Platforms

For in-depth analysis, the attack evaluation was carried out on a CAN bus testbed and on a test vehicle (ExpVehicle[1]). The attack code utilized $15kB$ of program storage and $1.5kB$ of dynamic memory. On the testbed, we used five nodes. Each node comprised an Arduino Uno board equipped with a CAN bus shield. One node acted as the attacker, and the other four emulated benign nodes.

[1]↑We decided to anonymize the make and model of our experimental vehicle since our research work tackles fundamental characteristics of CAN that are common to all CAN systems and not limited to this vehicle

All nodes were connected to a $500kbps$ CAN bus terminated with 120Ω on each end. For the vehicle, we used an Arduino Uno board equipped with a CAN bus shield as the attacker. We used the OBD-II port to connect directly to the CANH and CANL wires of the vehicle's high-speed CAN bus, which operates at a $500kbps$ baud rate.

3.5.2 Summary of Results

Network Mapping. To map the network, the attacker first makes a list of all the periodic message IDs on the bus and calculates the average period for each ID by recording the arrival times of N messages. We observed that certain low-priority messages have higher jitter components than higher-priority IDs, making their period length slightly change from one cycle to another, with a standard deviation of $\approx 0.6ms$. To account for such messages, we tried to pick an N that makes the error margin for the calculated period low enough to facilitate our source identification task. However, N represented a tradeoff, a high N lowered the error margin but increased the calculation time, and a small N decreased the calculation time but increased the error margin. To facilitate the use of preceded ID frame [20] in the source identification step, we wanted to keep the error margin around the length of an 8-byte message ($\approx 240\mu s$). Through a grid search, we found that $N = 20$ represented the optimum sample size and therefore used it.

On the testbed, the benign nodes were configured to transmit a total of 20 different benign message IDs with different periodicity ranging between $10ms$ and $100ms$. We identified all 20 different message IDs with correct periodicity in $\approx 9s$. Next, using Algorithm 3, we were able to identify all 4 transmitting ECUs and map all messages to their source ECUs with an accuracy of 100%. The mapping took $\approx 3mins$. On the vehicle, we identified all 50 periodic message IDs in $\approx 6mins$. The longest period was $5s$, while the shortest was $9ms$. Next, we were able to identify 4 transmitting ECUs on the bus and map all IDs to their sources with 100% accuracy, as shown in Table 3.1. The mapping took $\approx 9mins$.

Using Algorithm 3, we explain this time frame by noting that the overall mapping time $T_{map} = \sum_{ECU=1}^{4} T_{ECU}$. Here, T_{map} is the overall mapping time, and T_{ECU} is the time required to map a single ECU. For a single ECU, the majority of the time is spent in either pushing an ECU to the *error passive* state, checking an ECU's error state, or letting an ECU recover from the *error passive*

state (lines 10, 11, and 21). To push a message source to the *error passive* state or to check the state of a message source, we first observe an instance of the message, then intercept the next one (i.e., a total of 2 cycles). Additionally, following every check, we allow enough time t_{cool} for the long-period ID source to go back to $TEC = 0$. Finally, because of jitter, some messages require more than one attempt to be mapped (i.e., lines 19 and 20). Therefore, the time required to map one ECU becomes, $T_{ECU} = (2 * N_{\text{ids}} * (T_s + T_{avg})) + (t_{cool} * N_{\text{ids}}) + (T_{\text{jitter}})$. Here, N_{ids} is the number of unmapped IDs on the bus, T_s is the cycle length of the shortest-period ID, T_{avg} is the average cycle length of the unmapped IDs, and T_{jitter} is the time lost in failed mapping attempts. On our vehicle, $\approx 2mins$ were spent changing or checking error states, $\approx 3.8mins$ were the cool-off time, and $\approx 3.2mins$ were caused by jitter.

Victim Identification. We set up each ECU on the testbed to respond to a specific ID (per ECU). We were able to map each ECU's response to its respective ECU with 100% accuracy. On the vehicle and using OBD-II requests, we were able to identify the functions of the mapped ECUs by mapping OBD-II responses as described in Sec. 3.4.2. Specifically, using an OBD-II scanner, we identified four CAN IDS: $0x7E0$, $0x7E2$, $0x243$, and $0x241$, to which the Engine Control Module (ECM), Transmission Control Module (TCM), Electronic Brake Control Module (EBCM), and Body Control Module (BCM) responded, respectively. These ECUs responded at IDs: $0x7E8$, $0x7EA$, $0x543$, and $0x541$, respectively. The response time for each request was recorded and used to push the responder to the *error passive* state. Next, using the network map acquired in the network mapping stage, each of the responses was mapped to one of the transmitting ECUs as shown in Table 3.1. To the best of our knowledge, this is the first solution that could map triggerable, aperiodic messages with 100% accuracy without any special equipment.

Learning Victim's Recovery Behavior. On the testbed, two ECUs were set up to implement a fixed interval recovery model with a $35ms$ interval. Two other nodes were set up to implement the bare minimum model. We were able to learn the recovery models for all ECUs. Further, using SFBO, we were able to successfully suppress all nodes, one at a time, by attacking a single message, as explained in Sec. 3.3.1. For all ECUs, the optimum attack ID was found to be the ECU's message ID with the shortest period. On the vehicle, we successfully evaluated the SFBO technique on the

four mapped ECUs. To ensure the ECUs truly transitioned to the *bus off* state, we recorded the traffic after every attack and observed the lack of any IDs that belonged to the mapped ECU. This also validated our mapping results. The time recovery model for EBCM and BCM was identified as the *sequenced intervals*. For the TCM and ECM, we could not identify any pattern and hence treated them as following the *random* model. Additionally, for all ECUs, we were able to identify the optimum attack ID satisfying the two conditions mentioned in Sec. 3.4.3. Table 3.2 shows the optimum attack ID for each ECU. This evaluation is further detailed in Appendix A.1.

Table 3.2. Suppression rates for different ECUs on ExpVehicle.

ECU #	Function	Recovery Model	Optimum ID	S_{rate}
ECU-1	EBCM	Sequenced	0C1	97.5%
ECU-2	BCM	Sequenced	0F1	91.4%
ECU-3	TCM	Random	0F9	85%
ECU-4	ECM	Random	0C9	83%

Recovery Prevention. To assess the success of the attack, we define a metric called suppression rate (S_{rate}) that describes the percentage of time the victim is in the *bus off* state. Let t_{normal} and t_{attack} be a period of time when the attack is not running and when it is running, respectively. Also, let n_{normal} and n_{attack} be the number of target message IDs appearing on the bus during t_{normal} and t_{attack}, respectively. The suppression rate is calculated as $S_{rate} = ((n_{normal} - n_{attack})/n_{attack}) * 100$.

On the testbed, using the techniques described in Sec. 3.4.4, we were able to achieve an S_{rate} of 100% for at least $10s$ on all ECUs. After running the attack for 30 minutes, the average S_{rate} remained above 99.99%. On the vehicle, as shown in Table 3.2, using the techniques described in Sec. 3.4.4, we were able to achieve an average S_{rate} of 97.5%, 91.4%, 85%, and 83% for the EBCM, BCM, ECM, and TCM, respectively. The lower suppression rate on the vehicle, compared to the testbed, is due to the higher jitter in vehicular environments, leading the attacker to occasionally lose synchronization. This evaluation is further detailed in Appendix A.2

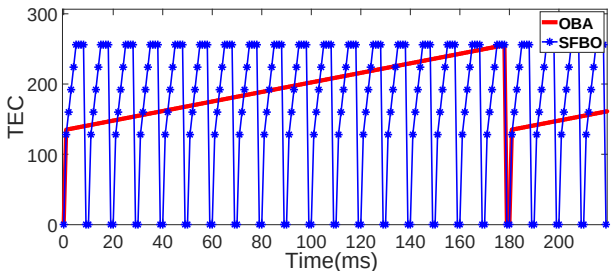

Figure 3.11. Swiftness of SFBO compared to the best case scenario of OBA.

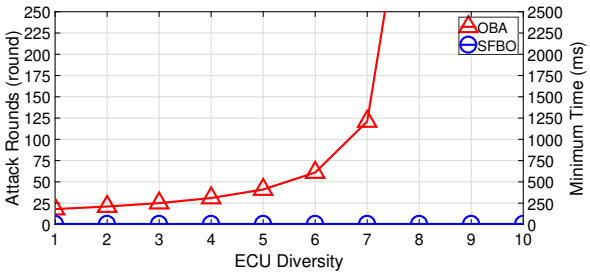

Figure 3.12. Illustrating the impossibility of OBA when ECU diversity exceeds 8.

Table 3.3. Comparison of suppression rates between OBA and SFBO in stage 3 and 4 of the STS attack.

ECU #	Message Periods (ms)	Recovery Model	# OBA Attack Rounds	OBA S_{rate}		SFBO S_{rate}
				Bus Load: 0%	Bus Load: 100%	All loads
ECU-1	10,20,50,90	Bare Min.	21	1.3%	13.2%	99.9%
ECU-2	10,20	Fixed	20	14.8%	14.8%	99.9%
ECU-3	10,50	Fixed	19	15.5%	15.5%	99.9%
ECU-4	10,20,50,100	Bare Min.	21	1.3%	13.2%	99.9%

3.5.3 Comparing SFBO to OBA

We compare the impact of using OBA [20] instead of SFBO in the third and fourth stages of STS. Specifically, we assess their impact on the suppression rate of STS. Additionally, we compare the feasibility of OBA and SFBO against ECUs transmitting multiple message IDs.

Swiftness. With SFBO, *only one attack round is required* to transition a node from the *error active* state to the *bus off* attack. Conversely, OBA required a minimum of 19 rounds of attacks, 1 round to transition the node from the *error active* state to the *error passive* state, and 18 attack rounds to transition it from the beginning of the *error passive* state to the *bus off* state. Essentially, crossing the *error passive* state into the *bus off* state previously represented the unresolved challenge in OBA. As shown in Fig. 3.11, in comparison to $\approx 5ms$ taken by SFBO, OBA required around $180ms$, making the fastest OBA attack 36 times slower than SFBO.

We note that 19 is the theoretical minimum number of rounds for OBA. In real-world cases, the number of rounds will be bigger. We assess the swiftness of SFBO and OBA on the testbed by measuring the time required to increase a victim's TEC from 0 to 256 for different ECUs. While SFBO pushed TEC to 256 in$\approx 5ms$ regardless of the ECU, OBA was ECU-dependant, taking 21, 20, 19 and 21 rounds, and $\approx$ 210, 200, 190 and $210ms$ for ECUs 1, 2, 3 and 4, respectively.

Impact on Suppression Rate (S_{rate}). To compare the impact of using OBA instead of SFBO on S_{rate}, we repeated stage 4 of the STS on the testbed under various loading conditions using OBA. While S_{rate} remained constant for nodes with a *fixed interval* model, regardless of the busload, the suppression rates changed for the ECUs that implemented a *bare minimum* model. This is because the busier the bus gets, the slower the instances of 11 recessive bits become, and the slower the node recovers. As shown in Table 3.3, while S_{rate} remained above 99.99% for SFBO, it ranged between 1.3% and 15.5% when using OBA.

ECU Diversity and Attack Feasibility. To explain why OBA requires more attack rounds with some ECUs, we note that OBA pushes the victim to the *bus off* state by launching rounds of attacks, each round increases TEC by 7. However, this is only true if the ECU sends one periodic ID. For ECUs sending multiple IDs, between one attack round and the next, other messages with different IDs will be transmitted, decreasing TEC by 1 with every transmission. This reduces the effective TEC change to less than 7 for each attack round, resulting in increasing the number of rounds required for the attack. As such, we define a metric named *ECU diversity*, which represents the ratio between the overall transmission rate of the entire ECU and the transmission rate of its fastest transmitting ID. The lowest diversity ratio is 1, which implies that the ECU only transmits one ID.

We compare the impact of the diversity ratio on the feasibility and swiftness of both SFBO and OBA. We set up an ECU to send multiple messages with different IDs. One ID was chosen as the target ID. The ECU increased its diversity ratio in steps from 1 (ECU only sends the target ID) to 10 (ECU sends the target ID along with 9 other IDs with the same period), and recorded the *minimum time* taken to push the ECU to the *bus off* state at each step, as well as the *minimum number* of attack rounds. As shown in Fig. 3.12, while the diversity ratio had no impact on SFBO, the time, and the number of rounds taken by OBA, increased exponentially. Most importantly, at a diversity

ratio of 8, the minimum attack time and the minimum number of attack rounds for OBA tended to infinity. *This means that OBA is impossible to launch against an ECU with a diversity ratio of* ≥ 8.

3.6 Responsible Disclosure

We reported the three discovered vulnerabilities to the Robert Bosch Product Security Incident Response Team (PSIRT). PSIRT acknowledged our work and offered to share details of the vulnerabilities with other automotive industry stakeholders. We also reported the vulnerabilities to the International Organization for Standardization (ISO). ISO referred us to the American National Standards Institute (ANSI), which directed us to the Society of Automotive Engineers (SAE). SAE acknowledged our contributions and submitted the vulnerabilities to a committee for review and consideration in the next revision. Finally, we reported the vulnerabilities to the Cybersecurity and Infrastructure Security Agency (CISA) through the CISA Coordinated Vulnerability Disclosure (CVD) process. CISA created a case for our report and asked us to report the vulnerabilities to Bosch and ISO, which we have done.

3.7 Defense Recommendations

With the vulnerabilities uncovered by CANOX being inherent to the CAN protocol, the fundamental defense causing no side effects is to revise the standard. However, noting that this may not be feasible, certain countermeasures may still be used in accordance with the current standard. Below, we present some possible mitigations and their potential downsides.

Passive Error Regeneration. Unfortunately, the only solution to stop an attack exploiting this vulnerability once it starts is to reset the ECU's CAN controller. Previous works have suggested this solution [32–34] to prevent DoS attacks. However, if the increase was happening due to legitimate errors, bringing a faulty CAN controller back to the *error active* state defeats the purpose of the fault confinement mechanism [35, 36], and may result in many performance issues. A possible solution is to reset only when an attack is suspected. This could be achieved by counting the number of errors in the passive error frames within a window. If the number exceeds a specific threshold, it could signify that the errors are due to an enforced passive error regeneration.

Deterministic Recovery Behavior. This vulnerability could be mitigated by clearing all transmission buffers upon entering the *bus off* state, or before re-entering the *error active* state.

Error State Outspokenness. CAN designers placed a *suspend transmission period* on successive transmissions and retransmissions in *passive* nodes to lower their priority. However, such a change could easily be spotted by an attacker. One countermeasure is to reset the CAN controller once it enters the *error passive* state. However, this may lead to performance issues. A better solution is for all ECUs to randomize the period between successive transmissions/retransmissions. For successive transmissions, this could be achieved by buffering the second message without marking it as ready for transmission until the random period elapses. For retransmissions, this could be achieved by disabling automatic retransmissions on the CAN controller and delegating this task to the application software. This helps conceal the *suspend transmission period* for *passive* nodes. However, it may also cause an increased overhead or priority inversions on the bus in some cases.

3.8 Discussion

Static Analysis of CAN Standard. CAN is not described in a formal language. As a result, attempts to analyze it for vulnerabilities using formal approaches, such as static analysis or model checking, require a tedious modeling process that often entails imprecision. Such imprecision could be caused by a number of reasons. For instance, abstract and vague parts of the standard could force the modeler to make assumptions that may not always reflect real implementation. Similarly, modeling a single component of the standard (e.g., error handling) ignores interactions between this and other components. An example of these two points is the *deterministic recovery behavior* vulnerability. Neither does the standard mention what to do with buffered messages when the node goes into the *bus off* state (vagueness) nor does it consider this issue as part of the error handling component. In contrast, CANOX speeds up this process and makes it more accurate by dynamically checking a real-world embodiment of the standard for vulnerabilities. Once a vulnerability is found, it is easy to check whether a standard or an implementation problem causes it.

Impact of Operating in Error Passive State. The *error passive* state was intended to offer a degree of protection against faulty nodes. Changing the error signaling method to transmit the *passive error frame* and reducing message priorities in certain scenarios, allowed CAN to operate in the presence of a faulty node. This also protected other nodes from engaging in a self-destructive behavior in the case of successive collisions. A good example of that is OBA, whereby by reducing the priority of the message retransmissions in the *error passive* state, the victim is able to break the time synchronization with the attacker. Hence, this protection slows down OBA, making it an ineffective DoS attack. However, CANOX reveals that these protections have an undiscovered, self-defeating side. Not being able to signal errors in a way that is apparent to all other nodes allows other nodes to step over *passive error frames*, generating a different kind of errors (form errors). This leads to the passive error regeneration vulnerability that an attacker can easily exploit to launch a swift DoS attack (SFBO) against a CAN node. Similarly, while priority reduction of an *error passive* node may offer some protection to CAN, it also reveals more information than necessary about the node. This leads to the error state outspokenness vulnerability, which can be exploited to identify the node's messages and by extension, map the network.

Other Uses of the Discovered Vulnerabilities. While we chose to present an advanced DoS attack to combine all the vulnerabilities into a single multi-staged attack, the discovered vulnerabilities could have other uses. For example, the source mapping technique described in Sec. 3.3.3 could be used for reverse engineering purposes. Similarly, recent works [19] have shown that an attacker may be able to impersonate a victim node on the CAN bus while evading intrusion detection systems (IDS) by being in the *error passive* state. Setting the victim's TEC as described in Sec. 3.3.1 comes in very handy for such a threat model. Furthermore, in systems where retransmissions are disabled, such as Time-Triggered CAN (TTCAN), the *passive error regeneration* vulnerability could be used to silently keep a node in the *error passive* state, causing a victim to miss deadlines, in case of successive transmissions, or allowing an attacker to inject messages in its place. Since the victim will not retransmit any failed messages or raise any *active error frames*, the injection may go undetected, especially if coupled with an IDS evasion technique such as the one just mentioned.

OBA vs. STS in Real World. The practical impact of the swiftness and persistence of STS is serious. For instance, STS is able to suppress the Electronic Brake Control Module (EBCM) continuously for $\approx 2.4s$ at a 100% suppression rate (97.5% over a 15-minute period). Consider a modern vehicle employing its adaptive cruise control mode and leaving a two-second-distance between itself and a vehicle ahead of it (2-second-rule). We can see that STS can completely disconnect the brakes long enough to cause the most serious consequence. Conversely, OBA will only suppress *one instance* of the brake message (in $\approx 0.5s$), and will not be able to follow this instance with persistent suppression. As such, OBA will result in an ineffective DoS attack allowing almost normal functionality of the brakes.

Limitation. STS causes packet collisions on the CAN bus. Hence, an IDS that monitors the number of collisions on the bus may suspect the presence of an attack. However, this does not affect the progress of the attack because of two reasons. (1) The first three stages of STS (i.e., the network mapping, victim identification, and recovery behavior determination) do not have to happen right before the final stage (i.e., the recovery prevention). They could take place in a "low and slow" manner over a period of time in order *not* to trigger the IDS. (2) Even if the attack gets detected, the attack cannot be stopped as it exploits inherent aspects of the CAN standard.

3.9 Related Work

Vulnerabilities of CAN. Prior research has demonstrated that after infiltrating CAN through a wired/wireless medium (e.g., USB, cellular, Bluetooth, and WiFi connections), an attacker can compromise an in-vehicle ECU node (e.g., telematics control unit) and execute arbitrary software codes on it [4–7]. Since CAN is devoid of any security features, the attacker can exploit the compromised node to launch a variety of attacks on other safety-critical nodes, which cannot be directly compromised [8]. Hence, it is imperative to develop frameworks that can methodically discover the full spectrum of vulnerabilities suffered by CAN under such scenarios [37]. To the best of our knowledge, CANOX is the first effort in systematically analyzing the error handling mechanism and discovering its security vulnerabilities.

ECU DoS Attack. Cho and Shin were the first to propose a DoS attack, referred to as OBA [20]. However, as shown here, OBA is incomparably slow in suppressing the victim and ineffective in stopping the victim's transmission persistently. As a consequence, it is unlikely to have a practical impact. Some other DoS attacks exploiting similar ideas as OBA required special hardware modules to launch the attack [21–23]. Hence, they required physical access to CAN, which makes them unscalable. Additionally, all the aforementioned solutions assumed the attacker already knows the ECU functions and the messages they transmit. In contrast, STS employs the discovered vulnerabilities to acquire this knowledge, then to rapidly and persistently suppress the victim using the existing abilities of a compromised ECU.

Network Mapping. Some prior works proposed using clock skews of ECUs to perform sender identification [24, 25]. However, their learning techniques were prone to inaccuracies and proved to be evadable [30]. Others suggested using voltage signatures of ECUs [26, 27] and hence required physical access. It is essential to note that all these solutions approached the issue from a defense standpoint. On the other hand, the severity of our technique lies in its ability to be used by a remote attacker. This is because it uses the existing ECU abilities to achieve the same task with higher accuracy. Additionally, we are the first to map aperiodic with existing ECU abilities.

3.10 Conclusion

We systemically analyzed CAN's error handling and fault confinement mechanism, focusing on operating in different error states, an understudied area in the CAN protocol. We built CANOX, a novel CAN testing tool to detect problematic behavioral changes across error states. CANOX uncovers three new vulnerabilities, which can be exploited by a compromised ECU to launch a multitude of attacks. We demonstrated the severity of the vulnerabilities by constructing a powerful attack, STS, in which an attacker with no knowledge of the vehicle's internals could map its internal network, identify ECU functions, shut down an ECU, and prevent it from recovering. We proved the attack's feasibility by evaluating it on both a CAN testbed and a real vehicle.

4. ZBCAN: A ZERO-BYTE CAN DEFENSE SYSTEM

4.1 Motivation

Modern vehicles contain hundreds of sensors and actuators, administered by Electronic Control Units (ECUs), including brake, engine, and steering control units. The most central communication channel among ECUs is CAN. Although reliable and robust against electromagnetic interference, CAN lacks any security measures. Researchers have demonstrated the feasibility of remotely compromising an ECU on the CAN bus [4–7, 13]. With the ever-increasing connectivity of today's vehicles, the ease of such compromises is expected to increase. Several works have shown that a compromised ECU can launch a plethora of attacks, including *message injection, impersonation, and flooding* [4–7, 13]. Moreover, recent works, including the first half of this dissertation, have unveiled vulnerabilities in CAN's error handling mechanism [19–23, 31, 38, 39]. These vulnerabilities allow attackers to deliberately inject collisions, map message sources, control the error states of certain ECUs or even persistently disable them [20, 23, 38].

To secure CAN traffic, two primary approaches have been proposed. One is the *cryptographic approach*, which relies heavily on cryptographic primitives (e.g., encryption, MACs, and hash functions). [40–51]. Unfortunately, this approach suffers from fundamental issues. The first is its *impact on performance* as cryptographic operations incur an unaffordable processing overhead for most commercial ECUs. Even worse, since the maximum payload length of a CAN message is $64\ bits$, these solutions are forced to either carve out a portion of an already-short message to attach authentication information, dropping the effective data rate, or use a completely different message, doubling the busload. Another issue is the *lack of intrusion confinement*. Since most of these solutions use group keys, if one ECU gets compromised, it can impersonate any node in the group. The last downside is the *lack of incremental deployability* or the ability to incrementally secure messages transmitted by a single ECU, without needing to update all ECUs at once.

The second approach is the *intrusion detection (IDS) approach*, which avoids the group-key problems and the computationally expensive cryptographic operations by delegating all security operations to a super-node that may have special equipment [26–29, 52–58]. This powerful node uses its abilities to detect traffic anomalies and flag them. However, this approach has its problems. First, IDSs take no measure to stop or prevent attacks. Second, most *CAN IDSs* do not achieve

single-message detection. Instead, they retrospectively detect *flows* of injected messages. This allows intermittent or gradual intrusions to pass unnoticed and contributes to these IDSs' inability to translate their attack detection into prevention, for a flow of messages is composed of a stream of individual messages. Being unable to determine whether an individual message is malicious or not prevents the IDS from taking action against any of the individual malicious messages that constitute the flow. This may render the IDS futile sometimes. For example, what benefit could a system gain by detecting the presence of a *flooding* attack if it cannot stop it? The mention of *flooding* attacks was only for the sake of argument. Unfortunately, the entire research field suffers from a *hyper-focus* phenomenon. A relatively vast amount of research has been dedicated to *message injection* and its variations in comparison with other attacks. This means that many attacks, such as flooding, or error handling attacks have very few defenses addressing them, let alone being effective. Finally, many defenses contradict one another and thus cannot be combined to protect against a more extensive attack-set. These problems have prevented any defense from being widely adopted.

We present Zero-Byte CAN, a versatile, low-overhead defense system that uses $zero\ bytes$ of the CAN message fields to protect against several attacks and offers *intrusion confinement*, *incremental deployability*, full *backward compatibility*, and *individual message guarantees*. This last feature allows ZBCAN to translate some of its *detection* abilities, into *prevention* ones, since individual malicious messages could be identified and stopped. ZBCAN does not use message fields, authentication messages, or computationally expensive operations such as encryption. Instead, it uses message timing alone to protect against the most common CAN attacks, including *injection, impersonation, fuzzing, flooding, collision injection, voltage corruption, and bus-off*.

ZBCAN is composed of a trusted *officer* node that can interrupt transmission and several software *agents*, installed on ECUs. The *officer* and each *agent* agree on a secret, endless, and dynamically generated sequence of inter-frame spaces, which the *officer* monitors for every message. The *officer* could be set to issue warnings, interrupt messages, or completely suspend violating nodes. Aside from attaching the *officer* to the bus, ZBCAN does not require any hardware changes. Further, since ZBCAN uses no message fields, it could be combined with solutions that do use them. For inclusivity, we evaluated ZBCAN's performance on a testbed using a real vehicle's data, its security and scalability on a testbed using artificial data, and finally, several security and performance aspects of ZBCAN on a real vehicle. Using ZBCAN, we achieved a *detection rate* of

Table 4.1. How ZBCAN compares with other CAN defense systems.

Defense Approach		Attacks						Features			Cost		
		Flood	**Injection**	**Replay**	**Collision Injection**	**Error Passive**	**Bus Off**	**Incremental Deployability**	**Single-Msg Detection**	**Intrusion Confinement**	**Modifies Message**	**Increases Busload**	**Processing Overhead**
Cryptographic 1‡		X	✓	✓	X	X	X	X	✓	X	X	✓	●
Cryptographic 2†		X	✓	✓	X	X	X	X	✓	X	✓	X	●
Voltage IDS	Detection	-	✓	✓	-	-	-	✓	X	-	X	X	○
	Prevention	X	X	X	X	X	X						
Frequency IDS	Detection	✓	✓	✓	-	-	-	✓	X	-	X	X	○
	Prevention	X	X	X	X	X	X						
Clk-Skew IDS	Detection	✓	✓	✓	-	-	-	✓	X	-	X	X	○
	Prevention	X	X	X	X	X	X						
ZBCAN	Detection	✓	✓	✓	✓	✓	✓	✓	✓	✓	X	X*	◔
	Prevention	✓	✓	✓	✓	✓	✓						

‡: Using extra messages to send authentication data.
†: Using message fields to send authentication data.
*: Sometimes, ZBCAN may cause a minute busload-increase (Sec. 4.4.3).

100% for *injection and replay attacks*, and *prevention rates* of 100%, 100%, 99.4%, 99.33%, and 98.5% for *error-passive, bus-off, collision injection, flooding, and injection attacks*, respectively. We summarize our contributions as follows:

- We present ZBCAN, a versatile defense system that uses inter-frame spaces to defend against the most common CAN attacks, offering both detection and prevention abilities.
- We introduce a new method to suspend any ECU as soon as it starts transmitting a frame called *Instant Bus-Off*. This method could be used to suspend intruding nodes.
- We offer worst-case response time analysis to systems with ZBCAN. We apply our analysis on a real CAN bus and show that all messages are guaranteed to be schedulable.
- We offer a probabilistic security analysis of ZBCAN against different attack types.
- To show its applicability, we evaluate different aspects of our system on a CAN testbed, on a real vehicle's traffic, and directly on a real vehicle's CAN bus.

4.2 Related Work

Intrusion Detection Approach. To avoid using cryptography, researchers proposed using lightweight Intrusion Detection Systems. Some IDSs rely on traffic features such as message frequencies, lengths, payloads, or clock skews to detect anomalies [24, 52, 54–57]. Others use physical features such

as the unique electrical characteristics of each ECU, manifesting in their transmission voltage levels [26–29, 58]. Nevertheless, IDSs have their problems. Namely, many of these systems were shown to be evadable [19, 30]. Further, despite the high detection rates they present, most of them do not detect single injections, but flows of N injections, leaving room for low-level attacks to pass unnoticed and preventing them from translating their detection abilities into prevention.

Timing-Based Approach. INCANTA [59] proposed adding secret delays to the expected arrival times of periodic messages, with receivers inspecting the delay of every message. However, the accuracy of such delays degraded significantly for lower-priority IDs. CANTO [60] suggested pre-scheduling bus traffic to avoid unexpected delays of lower priority messages. Unfortunately, both methods use up to 8 message bits and incur processing overhead on the receiving side. Other works [61, 62] used similar techniques with variations such as using authentication messages, a monitor node, the delays between each message and its authentication message, or using multiple covert channels. Similar to INCANTA and CANTO they did not eliminate using frame bits or authentication messages. Additionally, all the aforementioned solutions use a *primitive form of delay* that is vulnerable to an attacker purposely injecting higher priority messages and causing a target-message to be late, do not offer prevention, focus only on *injection*, and work only for periodic messages. We propose *a different kind of timing channel*, based on *inter-frame spacing*, which cannot be tampered with and works for periodic as well as aperiodic messages. We use it to defend against an extensive set of attacks and offer both detection and prevention, *without* using any message fields or sending extra authentication messages.

Other Approaches. Few works have addressed attacks other than *injection*. Namely, to prevent bus flooding, researchers suggested modifying the network's hardware to allow for the isolation of attackers[63, 64]. Such solutions are very expensive to implement due to their requiring extensive changes to the network hardware and architecture. Other works suggested manipulating the ID to bypass targeted flooding [65, 66] or randomizing portions of it to prevent error handling attacks[19]. Nonetheless, these systems cannot be deployed where IDs are used to convey commands, responses, or anything beyond their usage as mere identifiers as in most diagnostic protocols.

4.3 ZBCAN

4.3.1 Architecture and Operation Overview

As shown in Fig. 4.2, ZBCAN consists of a central monitor node, able to stop messages during transmission, called the *officer*, and a set of software *agents*, installed on every ECU. Each ECU privately agrees on a secret, non-repeating, and unique sequence of inter-frame spaces, called the *In BetweeNs* ($IBNs$), with the *officer*, and then enforces these sequences upon outgoing messages. If the *officer* detects a message with the wrong IBN, or an unknown ID, it stops it right after the ID portion of the message, thus preventing the message from being received by any ECU. Depending on the *officer's* setting, it may ignore the message, issue a warning, stop it, or disable its transmitter (Sec. 4.9). This way, several attacks could be prevented at once. Namely, *error handling attacks* rely on a technique called *simultaneous transmission* (Sec. 2), where attackers have to send a message exactly at the same time their victim transmits. With ZBCAN, they need to guess the exact IBN value for every message to transmit simultaneously. Similarly, *injection* and *flooding* attackers need to guess the correct IBN value for every message. Otherwise, their messages will be detected by the *officer* due to their wrong IBN and hence stopped.

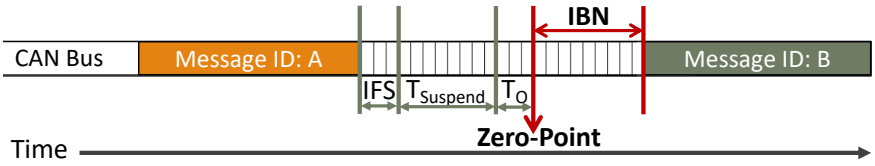

Figure 4.1. IBN basic concept.

The In BetweeN (IBN) As shown in Fig. 4.1, we use the term (IBN) to refer to the spacing between any two consecutive frames, measured from a *zero-point* (explained in Sec. 4.3.2). Although this definition is similar to that of the *Inter-Frame Spacing* (IFS), in most definitions, IFS refers specifically to the three bits following the end of a frame. To avoid confusion, we use the term IBN. Per the standard [67], CAN controllers initiate transmission after sensing the bus idle. This happens by sampling voltage at time intervals equal to one-bit each. As a result, the spacing between the end of one frame and the beginning of another is not continuous but discrete, meaning, *it is a multiple of a bit's length*, plus a small extra delay caused by clock skews and propagation delay. Therefore, if

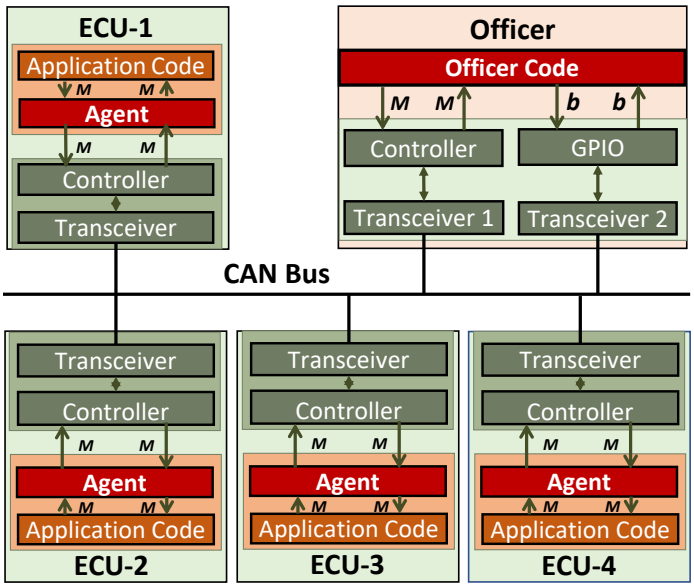

Figure 4.2. Architecture of a system implementing ZBCAN. Symbol (M) refers to messages. Symbol (b) refers to bits.

we can find ways to ignore this small delay, as explained in Sec. 4.4.3, we can view the spacing as discrete and measure it using bit-length units or simply $bits$.

IBN Sequence. To illustrate how to use IBN as a signature, assume that a generic message is currently being transmitted on the bus. Further, assume messages of $ID = X$ have a sequence of endless, secret, and non-repeating IBN values to be followed. As shown in Fig. 4.3, when wishing to transmit a new instance of X, the $agent$ of X waits until the ongoing generic message transmission concludes, counts a distance equal to the scheduled IBN (IBN_{sc}) in the sequence, then transmit. The *agent* does not wait until $IBN \geq IBN_{sc}$ but *exactly* $= IBN_{sc}$.

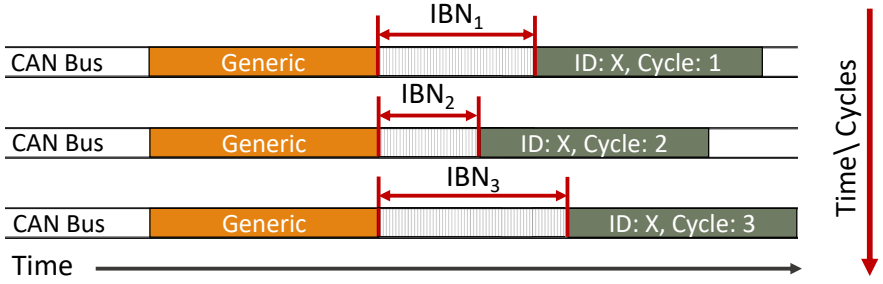

Figure 4.3. A running IBN sequence as a message ID signature.

Officer. As shown in Fig. 4.2, the *officer* is a trusted node that has the ability to securely store keys and has access to the bus through two channels, one through a CAN controller, and another directly through a GPIO and a CAN transceiver. The GPIO channel serves three purposes: (1) accurately measuring the IBN of every message, (2) reading message IDs before their data is delivered, and (3) allowing the *officer* to inject *error frames* on demand to stop any message. The *officer* is connected to the CAN bus in parallel as other nodes. It is not a gate or a bottleneck and causes no delay to messages. Instead, it acts as an observer who can immediately intervene. Its role is to monitor the enforcement of the IBN sequence. If it detects a message violating its IBN sequence or a message ID that is not allowed, it stops it before being received by any ECU. To be able to do so, the *officer* knows all the allowed IDs on the bus and their secret sequences.

Agent. The *agent* is a software installed on every ECU we wish to protect. It does not require any hardware changes to the ECU. The *agent's* role is to apply the IBN sequence upon outgoing messages. This sequence is unique per message ID and shared between two parties only: a ZBCAN *agent*, and the ZBCAN *officer*. One *agent* does not know, and hence cannot mimic, the sequences of any other *agent*, even if it gets compromised. As shown in Fig. 4.4, the agent is composed of four blocks: a Start Of Frame (SOF) ISR, an End Of Frame (or receive/transmit/error) (EOF) ISR, a Timer ISR, and a *buffering and* IBN *sequence extension* library.

Message Reception. Upon reception, $agents$ do not perform any computations. If a message is received successfully, it means that the *officer* has checked it, verified its IBN, and decided not to interrupt it. This way, we eliminate any processing overhead on the receiving side.

4.3.2 IBN Implementation Details

Zero-Point Calibration. The *zero-point* cannot be the same as last frame's last IFS bit for two reasons. First, nodes operating in the *error-passive state* have an additional 8-bit suspend-transmission penalty ($T_{Suspend}$), enforced at the protocol controller's level. If IBN_{sc} is 0, and the *zero-point* is the last IFS bit, an *error-passive* node will violate this value. Second, if IBN_{sc} is too low, an ECU with low computational power may not have enough time to initiate transmission

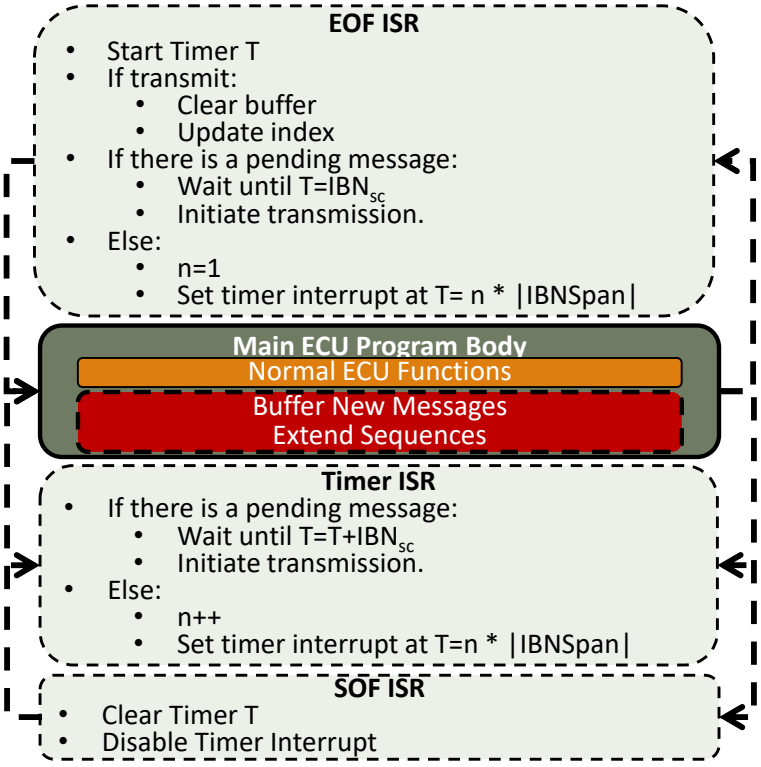

Figure 4.4. Agent components (dashed) within an ECU.

in time but after an overhead period (T_O). T_O should be measured empirically for every system. Accordingly, as shown in Fig. 4.1, we set $zero\text{-}point \geq IFS + T_{Suspend} + T_O$.

Measuring T_O. *Agents* are composed of a library and three Interrupt Service Routines (ISRs) (Fig. 4.4). T_O is dependent on the end of frame (EOF) ISR, which is responsible for clearing the transmit buffer, updating the sequence index, checking if there are pending messages to be transmitted, then applying the IBN value at the next transmission. Ideally, the ISR should be able to execute these tasks by the end of $T_{Suspend}$ in Fig. 4.1. However, some ECUs may take longer. The effective ISR processing time for an ECU could be measured during the system design phase by sending a test message at the end of the EOF ISR (without any IBN), then measuring the distance between the last message on the bus and the test message. The system's T_O, should be set to the *longest* ISR processing time of any ECU.

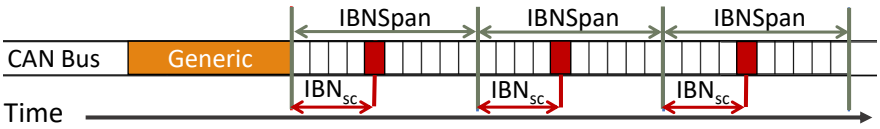

Figure 4.5. Dividing the timeline into distances =‖IBNSpan‖ allows for using Modulo IBN instead of Absolute IBN.

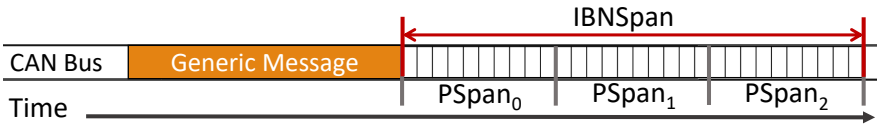

Figure 4.6. Dividing IBNSpan into exclusive priority spans.

IBNSpan. If the bus is busy and IBN_{sc} is too long, the *agent* may never find the opportunity to transmit. To prevent this, all IBN values should be kept within a span ($IBNSpan$) so that any message with $IBN_{sc} \in IBNSpan$ is guaranteed to transmit within a window not exceeding its deadline (explained in Sec. 4.4.2). We use the notation $\|IBNSpan\|$ to refer to the number of elements (IBN values) in the $IBNSpan$ set.

Modulo IBN. Since IBN is counted from the last frame on the bus, if a message is generated during a long idle period, it will have to wait until a message appears. Even worse, if all ECUs are also waiting for a message to appear, no messages will transmit. To prevent such problems, starting at the last *zero-point*, we divide the timeline into slots of length $= \|IBNSpan\|$. Instead of having to send the message only at a spacing $= IBN_{sc}$, we send it at any spacing (d) that satisfies the condition: $d \bmod \|IBNSpan\| = IBN_{sc}$. This way, if a message is generated in an idle period, it waits until the beginning of the next $IBNSpan$, counts a number of bits $= IBN_{sc}$, then initiates transmission, as shown in Fig. 4.5. Beginning from here, the term IBN refers to *Modulo IBN*.

Priority and IBN. Without ZBCAN, if two messages with IDs 0 and 10 are pending transmission at the same time, they will both go through an arbitration phase that ends in $ID:0$ winning and transmitting first. With ZBCAN, if $ID:0$'s scheduled IBN is 10 b, while $ID:10$'s scheduled IBN is 0 b, $ID:10$ will transmit first, inverting the priority system.

To guarantee that such a scenario does not cause timing deadline violations for time-sensitive messages, we enforce our own priority system. First, we divide $IBNSpan$ further into N_{pri} non-overlapping ranges called *priority spans* ($PSpans$), each representing one priority level as shown in Fig. 4.6. Next, we arrange all message IDs in ascending order, based on their deadlines, then group them into $N_{pri} \geq 1$ priority groups (P_{group}s). Each P_{group} contains one or more message IDs sharing the same $PSpan_{group}$, where $PSpan_{group} \in IBNSpan$. P_0 is dedicated $PSpan_0$ and contains the IDs with the shortest deadlines, while $P_{N_{pri}-1}$ is dedicated $PSpan_{N_{pri}-1}$ and contains IDs with the longest deadlines. This way, we guarantee that messages with the shortest deadlines have a higher priority. In Sec. 4.4.1, we model the worst-case response time (WCRT) for messages in a ZBCAN system, then use this model to map message IDs into *priority groups* and guarantee that time-sensitive messages arrive in time.

Dummy Messages. To prevent IBN inaccuracies due to the inherent clock skew among ECUs, *dummy* messages of zero-byte length may need to be inserted after long idle periods to force all ECUs to resynchronize. We detail this in Sec. 4.4.3.

4.3.3 Operation Implementation Details

Registration and Sequence Exchange. Each *agent* starts its operation by an exchange with the *officer* to establish the first sequence for each ID. This exchange happens only at the beginning of operation. Each *agent* has a secret key, *pre-shared only with the officer*. *Agents* do not know each others' keys. However, the *officer* knows the *pre-shared keys* of all *agents*. The details of pre-sharing this data are outside the scope of this discussion. Using these keys, each *agent* starts its operation by securely and randomly generating a seed SR and then exchanging it with the *officer*. Both the *agent* and *officer* use the seed, the *pre-shared key*, and an agreed-upon *pseudo-random function (PRF)* to generate a *session key*. Next, based on the number of IDs per ECU ($Nids_{ecu}$), both the *agent* and *officer* generate $Nids_{ecu}$ seeds, each separated by an agreed-upon offset Off. The first ID's seed is $= SR + Off$ and the last ID's seed is $= SR + (Off * Nids_{ecu})$. Finally, using the seed, *session key*, and PRF, we generate a number of length $SeqLength$. This number, per ID, will act as the ID's first IBN sequence (Seq_{id}), from which individual IBN values are drawn.

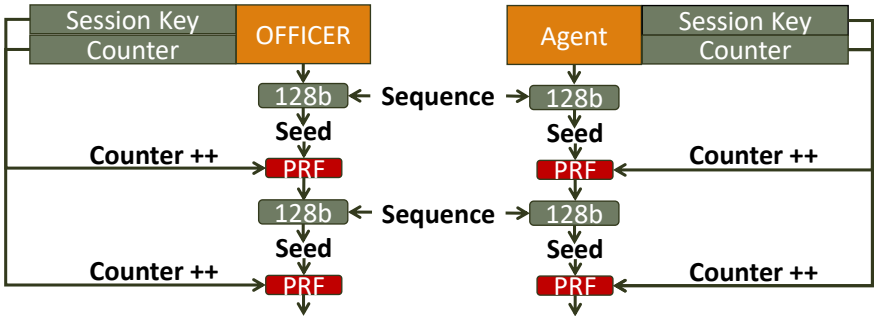

Figure 4.7. Extending a 128b sequence.

Sequence Usage. Every $agent$ keeps an $index_{\text{id}}$ for each ID's IBN sequence. With every transmitted message, the *agent* consumes the bits pointed at by $index_{\text{id}}$ from the sequence and then increments $index_{\text{id}}$. Specifically, every transmission, the *agent* extracts $\log_2 \|PSpan_{group}\|$ $bits$ from Seq_{id}. The value of the bits act as the scheduled IBN value for the next message. For example, if a message belongs to a *priority group* whose $PSpan_{group} = [32, 63]$, then $\|PSpan_{group}\| = 32$, whose $\log_2$ is 5. If we extract the 5 $bits$, pointed at by $index_{\text{id}}$, and find their value $= 15$, then $IBN_{sc} = 32 + 15 = 47$ $bits$. Once transmission is initiated at bit 47, we increment $index_{\text{id}}$.

Sequence Extension. After the initial sequence exchange, each sequence could be used to generate new sequences throughout the operation without having to re-exchange sequences with the *officer*. We call this operation *sequence extension*. To keep a running sequence, we recommend using a fast PRF. As shown in Fig. 4.7, the *agent and officer* start with a *session key* and a different seed per ID. Using the PRF, they generate an initial sequence of length $SeqLength$ for each ID and start drawing bits from it with every transmission. A circular buffer holding two sequences (a current one and a future one) should be kept to avoid interruptions. Once a sequence is consumed, a new one should be extended to replace it. We also recommend using a counter of length $SeqLength$, to be incremented and *XORed* with the session key with every extension for augmented security. $SeqLength$ needs to be small enough for a limited-space ECU to be able to store it. For a typical configuration of $SeqLength = 128b$ and $\|IBNSpan\| = 64b$, each transmitted message draws $6b$, an extension happens every $128/6 \approx 21$ messages.

Sequence Exchange Frequency. Without counters, sequences could be extended until the result of one extension operation repeats or equals the initial seed. If that happens, all the following sequences will also repeat. For a $128b$ sequence, the probability of an extension generating such a number is very low ($1/2^{128}$ with every extension). Further, since we use counters, both the output and the counter values need to be the same as a previous entire combination for sequences to start repeating. The probability of that is even lower ($1/2^{256}$).

For an *agent* with 16 IDs, if we do not want two IDs to have the same counter values for stricter security, we can divide the counter values into exclusive ranges for each ID and only extend the sequence until the counter reaches the end of its range ($2^{128}/16\ extensions$). Assuming an extension covers $21\ messages$, then for a fast transmitting ID, with a $10\ ms$ period, we could perform sequence extensions for $21 * 10 * 2^{128}/16 = 4.46 * 10^{39}\ ms$, before a counter repeats. Alternatively, *agents* could perform an exchange once at the beginning of operation.

Officer Policing. If the *officer* detects a message with a wrong IBN, it interrupts it using an *active error* right after reading its ID field. This prevents its payload from appearing on the bus or being consumed by any receiving ECU. Right after the interruption, it issues a *warning and resynchronization* message with an ID = ID_{warn}, a system parameter. Only the *officer* is allowed to send this ID. The message contains the violating ID and the $\mathrm{index_{id}}$ of the next expected IBN. Upon reception of a warning message, ECUs read which ID violated its sequence. If an ECU is a transmitter of the violating ID, it updates its $\mathrm{index_{id}}$ to the one in the message. If it is a receiver, it takes note of the possibility of the data being compromised. After N_{warn} successive warnings or if the message has a prohibited or unknown ID, the *officer* suspends the intruding node (Sec. 4.3.4).

Errors. If an *agent* encounters an error while transmitting a message, it should increment its $\mathrm{index_{id}}$. This is due to the fact that most errors happen after the ID portion of a message, meaning, after the *officer* has witnessed and approved the ID and IBN.

Queuing. Since we group every message on the bus into N_{pri} *priority groups* and consider all IDs within a group to have the same priority, we recommend that within each *agent*, messages in the same *priority group* share a FIFO queue as shown in Fig. 4.8.

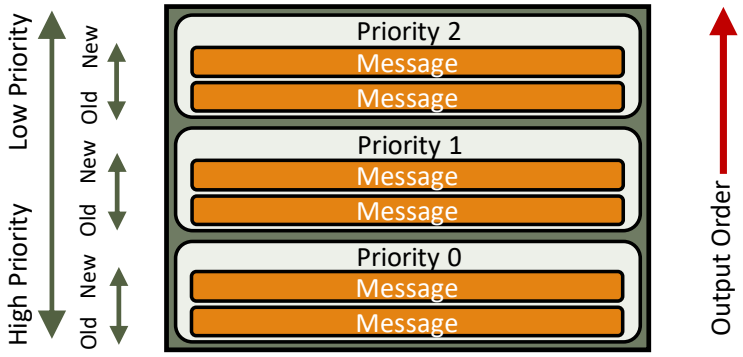

Figure 4.8. Priority FIFO.

4.3.4 Disabling Transmitter (Instant Bus-Off)

We propose a new technique to push an ECU to the *bus-off state* by targeting a single message. It requires equipment that is able to accurately inject individual bits directly into the bus. Only the *officer* could do that per our threat model (Sec. 2.3). The method is as follows. (1) We pick a frame on the bus and wait until a one is being transmitted. Once that happens, we inject a zero. (2) After a single bit, the transmitter detects this error and attempts to send an error frame composed of 6 zeros (flag) and 8 ones (delimiter). (3) After the delimiter starts, we release the bus for a single bit, allowing the one to appear. (4) After the one, we re-inject a zero. Consequently, step (2) repeats. We repeat steps $(1-4)$ 32 times, where the transmitter enters the *bus-off state* as illustrated in Fig. 4.9. This process could take as little time as $(7*32)+(1*31)=255\ b$ ($510\ \mu s$ on a $500\ kbps$ CAN bus) to transition a node from the *error active* state to the *bus off* state.

Although in the first half of this dissertation [38], we proposed the *single-frame bus-off* (SFBO), this technique outperforms it in two ways: (1) *SFBO* requires $\approx 5\ ms$. Our technique requires 512 μs, up to $\approx 10X$ faster. (2) *SFBO* requires automatic re-transmissions to be enabled. As such, an ECU could protect itself by disabling automatic re-transmissions. Our technique does not rely on re-transmissions and hence *cannot be escaped* once launched.

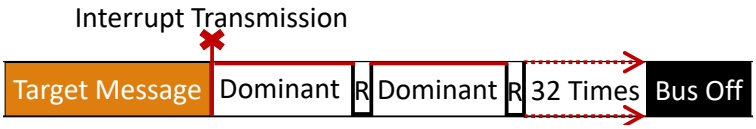

Figure 4.9. Successively interrupting error frame delimiters 32 times instantly pushes transmitters to the bus-off state.

4.4 Performance Analysis

4.4.1 Worst-Case Response Time Analysis

Several works [68–74] have analyzed the Worst-Case Response Time (WCRT) in CAN systems. We use the findings of [69] as a starting point. In Equation 4.1, R_m refers to the WCRT of a message, J_m refers to the queuing jitter or the longest time between initiating queuing and actually queuing a message, w_m refers to the queuing delay or the maximum time a message could wait in the queue before initiating transmission, and C_m refers to the longest transmission time of message m. Every message ID m should have two metrics defined: (1) T_m to represent the period of a periodic message or the minimum inter-arrival time between two instances of an aperiodic message, and (2) D_m to represent the timing deadline or the maximum allowed delay for the message. A message is *schedulable* only if $R_m \leq D_m$, or if its worst-case response time is smaller than its timing deadline.

$$R_m = J_m + w_m + C_m \tag{4.1}$$

To calculate the worst case queuing delay w_m in Equation 4.1, we use Equation 4.2. B_m refers to the blocking delay or the time message m could wait for a lower priority message, currently in transmission, to conclude. T_k refers to the minimum time-interval between successive launches of the queuing task of message k, and τ_{bit} to the bit-time.

$$w_m^{n+1} = max(B_m, C_m) + \sum_{\forall k \in hp(m)} \lceil \frac{w_m^n + J_k + \tau_{bit}}{T_k} \rceil C_k \tag{4.2}$$

With ZBCAN, messages wait IBN_{sc} before transmission. The effective transmission time for message m then could be viewed as $IBN_{sc,m} + C_m$. Assuming $m \in$ *priority group* (P_n) and $PSpan_n$ starts at point $(Span_{n,beg})$, the maximum $IBN_{sc,m}$ message m could wait is $G_{max,n} =$

$\|PSpan_n\| - 1 + Span_{n,beg}$. As a result, we define $EC_m = G_{max,n} + C_m$ to represent the *effective maximum transmission time* of m, or the maximum time it could wait *if the bus is available* plus its actual maximum transmission time C_m. Similarly, the *effective maximum blocking delay* EB_m includes the lower priority message's waiting time. Since message m shares the same priority level with a whole group, defining which IDs have higher priorities within the group becomes difficult. The worst case is for the longest message to be currently in transmission, for all messages of higher-priority and same-priority groups $hp(m)$ and $sp(m)$, to be pending transmission at the same time, for all higher-priority messages to receive the maximum IBN value for their $PSpans$, for all messages of the same group, including message m to wait for $G_{max,n}$, and for m to lose arbitration to every message and transmit last. Thus, we deduce that the worst queuing delay for our system could be represented by Equation 4.3.

$$w_m^{n+1} = max(EB_m, EC_m) + \sum_{\forall k \in hp(m) \cup sp(m)} \lceil \frac{w_m^n + J_k + \tau_{bit}}{T_k} \rceil EC_k \tag{4.3}$$

This finding is similar to that of [70] for adjacent priority FIFOs, with the term f, denoting the buffering delay, set to $zero$, since we assume that ECUs are implementing *priority FIFOs*.

Finally, Equations 4.1 and 4.3, give us insight on what factors influence the WCRT of a message. We expect for factors such as the bit-time (baud-rate), message length, jitter, number of messages and their inter-arrival times, number of priority groups, and $\|PSpan\|$ to have a direct influence. We also expect for messages in the higher priority groups to have a higher influence on the WCRTs of the system than lower priority groups.

4.4.2 Priority Grouping

We define the ratio $Ratio_{safe} = R_m/D_m$ to represent the $WCRT$ of a message m divided by its deadline. To guarantee that time-sensitive messages will not violate their deadlines (*schedulable*), we map different message IDs to different *priority groups* such that the condition: $Ratio_{safe} \leq 1$ holds for all messages and groups. Grouping should take place during the system design phase and not during operation. To optimize our grouping, we define two objectives. The first is to minimize $Ratio_{safe}$. Assuming the system has a fixed $IBNSpan$, then it is obvious that the security of the system drops with every division of this span. Hence, the second objective is to minimize the number

Algorithm 4 Priority Grouping Algorithm

```
AllIDs ← System ID list
n ← 0
Ratio_safe ← 0
while Ratio_safe ≤ 1 do
    System ← Schedulable
    while AllIDs != Empty && System != Unschedulable do
        Create new group P_n
        while P_n != Full && AllIDs != Empty do
            Add ID to P_n
            Remove ID from AllIDs
            if !( All P_n IDs are safe) then
                Remove ID from P_n
                Add ID back to AllIDs
                if P_n Empty then
                    System ← Unschedulable
                    Return IDs from all groups to AllIDs
                P_n ← Full
                n++
    Ratio_safe = Ratio_safe + 0.05
if Ratio_safe > 1 && AllIDs != Empty then
    System ← Unschedulable
```

of priority groups. Alg. 4 illustrates how to achieve these objectives. In the algorithm, the term "safe" means $Ratio_{safe} \leq 1$ or that the worst-case delay is shorter than the deadline.

4.4.3 Discretizing IBN Challenges

ZBCAN works by discretizing the spacing between consecutive frames, then controlling this spacing (IBN) to achieve its security goals. Nonetheless, the impact of factors such as the *propagation delay* on this "discretization process" should be studied to prevent any IBN inaccuracies.

Propagation Delay. Controllers read the value of a bit by taking voltage samples from the bus at bit-lengthed intervals. A bit-time is divided into four segments. The sample point is configurable, and is taken between the third and fourth [67]. For the *propagation delay* to cause IBN inaccuracies, it needs to exceed the sampling point. Assuming a typical propagation delay of $5\ ns$ [75], a baud rate of $500\ kbps$, and an *officer's* sampling point of 65%, the *round trip propagation delay* needs to exceed $1300\ ns$ (cable length of $> 130\ m$) to constitute a problem. For a typical CAN bus with a 2 to $15\ m$ length, this problem is irrelevant.

Clock Skew and Dummy Messages. Due to the inherent clock skews among ECUs, they gradually lose synch. To counter that, CAN requires all controllers to re-synchronize at the rising edge of every new frame's SOF. In ZBCAN, if the clock skew of one ECU causes it to start transmitting past the sampling point of the officer, it will cause a *false positive*. To prevent that, we take advantage of CAN's re-synchronization mechanism. Specifically, we define the metric d_{skew} that refers to the minimum spacing between messages that causes any ECU's clock skew to start causing IBN inaccuracies. By inserting a dummy message of *zero-byte* length at an idle spacing $d_{dummy} < d_{skew}$, all clocks re-synchronize before the clock skew causes inaccuracies.

4.4.4 Overhead Analysis

A *sequence extension* operation for a specific ID happens every $L_{seq}/\log_2(\|PSpan\|)\ messages$. On the *officer's* side, the sequences are extended for every message ID in the system. However, On the *agent's* side, sequences are extended only for the message IDs that the *agent* transmits.

To function properly, the *agents* and the *officer* need a minimum amount of memory to hold variables such as keys, sequences, etc. Specifically, *agents* require at least $(2 * L_{sequence}) + ((3 * L_{sequence}) + L_{index}) * N_{ids-agent}\ bits$, where $L_{sequence}$ refers to the length of the sequence, L_{index} to the length of the index, and $N_{ids-agent}$ to the number of IDs that the *agent* sends. Similarly, the *officer* requires at least $(2 * L_{sequence} * N_{agents}) + ((3 * L_{sequence}) + L_{index}) * N_{ids-system}\ bits$, where $N_{ids-sys}$ and N_{agents} refer to the number of IDs and number of *agents* in the system, respectively. A more detailed analysis of the memory and processing overhead is provided in Appendix C.1.

4.5 Security Analysis

Compromised Agent Abilities. Our threat model (Sec. 2.3) assumes a remote attacker that has all the information of an *agent* but is limited by the ECU's hardware. This is equivalent to the attacker having full control over all OSI layers except for the physical and data link layers. While ECUs communicate on the CAN bus through a CAN controller, the *officer* can connect directly to the bus and could manipulate the data link layer. Further, an *agent* knows only its pre-shared key (Sec. 4.3.1), which is used to agree on an IBN sequence. The *officer*, on the other hand, knows the pre-shared keys of all *agents*. Consequently, a compromised *agent* cannot read message IDs during

transmission, stop messages on demand, alter protocol rules, or establish IBN sequence agreement with other *agents*, since it does not have the necessary hardware or keys. This limits its abilities to receiving and transmitting full messages and controlling their IBN values, which could be used to push another ECU's messages off sequence, launch message injection (including impersonation, masquerade, etc.), error handling, or flooding attacks. Here, we discuss these attacks.

4.5.1 Off Sequence Attack

In ZBCAN, each message ID follows a strict IBN sequence. If an attacker is able to guess the scheduled IBN for a target ID once, the *officer* will update its $\mathrm{index}_{\mathrm{id}}$ but the legitimate transmitter will not. Consequently, when it sends its next instance of the message with $IBN = IBN_{sc}$, it will be *off-sequence*, since the *officer* will be expecting an $IBN = IBN_{sc+1}$. However, since each ID has its own sequence, even if one ID is pushed off sequence, the *agent* will be able to transmit the rest of the IDs normally. Further, since the *officer* will stop the first message with unexpected IBN then issue a *warning and resynchronization* message containing the expected $\mathrm{index}_{\mathrm{id}}$, only the legitimate *agent* will be able to synchronize since it only it has the IBN sequence.

This method turns a security weakness into a strength, guaranteeing that injections will be detected in 100% of cases since even a successful injection always results in the legitimate ECU going off sequence and the *officer* issuing a warning message to all receivers. Further, it is better than having each *agent* monitor whether its messages are being impersonated and then automatically resynchronizing in terms of the performance overhead incurred, since *agents* have to only watch for messages with ID_{warn}, instead of every ID they transmit.

4.5.2 Injection and Detection Window

Attackers could inject messages with IDs that exist in the network (e.g., masquerade and impersonation) or random IDs that do not necessarily exist (e.g., fuzzing attacks). Assuming a smart attacker who wants to target a specific ID and who knows the system IDs, their groups, and $PSpan$, ZBCAN offers three probabilistic security guarantees for injection attacks: *Individual-Message Detection*, *Individual-Message Prevention*, and *Flow Detection*.

Individual-Message Detection. Assume a periodic message m with a period T belonging to a *priority group* with a $\|PSpan\| = n$ and a scheduled $IBN = IBN_{sc}$. The probability of guessing IBN_{sc} is $1/n$. Within a time period $\leq T$, the legitimate ECU will send its message with the same $IBN = IBN_{sc}$. Since the *officer* will be expecting an $IBN = IBN_{sc+1}$, the injection will be detected within a time window $\leq T$, except if IBN_{sc+1} is randomly $= IBN_{sc}$. Since the sequence is generated using a PRF, the probability of IBN_{sc+1} being equal to IBN_{sc} is also $1/n$. To generalize, let the *detection window* (w) be an integer representing the number of periods/cycles of duration T since the injection, the probability of detecting an injection within a window w is represented by Equation 4.4. Note that $P(w)_{det}$ always tends to 1 given enough cycles.

$$P(w)_{det} = 1 - \frac{1}{\|PSpan\|^{w+1}} \tag{4.4}$$

Individual-Message Prevention. To prevent an injection, the *officer* needs to detect it as soon as it appears on the bus, and before it is delivered to the receivers. This means that the *probability of prevention* $P_{prevent} = P(0)_{det}$, as shown below:

$$P_{prevent} = 1 - \frac{1}{\|PSpan\|} \tag{4.5}$$

The expected number of trials before a successful injection is $E(\text{inj}) = \|PSpan\|$, not $\|PSpan\|/2$, since the *officer* increments $index_{\text{id}}$ for the sequence with every observed ID instance whether its IBN is accurate or not.

Injection Flow Detection. For a flow of f messages not to be detected, every single message in the flow should pass unnoticed. In other words, an injection flow is detected when any of its messages are detected. Equation 4.4 could be generalized to quantify this probability to become:

$$P(w, f)_{det} = 1 - \frac{1}{\|PSpan\|^{w+f}} \tag{4.6}$$

Random Injections. ZBCAN allows a message to transmit if and only if the message ID is allowed on the bus and the message is following its Seq_{id}. Since random injections violate both conditions, their rate of prevention and detection is $\approx 100\%$.

4.5.3 Error Handling Attacks

Collision Injection. To inject a collision using *simultaneous transmission* (Sec. 2), the attacker needs to estimate the transmission time of the victim message, send a *synch message* slightly before its expected arrival, followed by a message of the same ID. With ZBCAN, the attacker cannot randomly inject this high-priority message for synchronization or it will be stopped by the *officer*. Further, the attacker has to accurately guess the scheduled IBN for the victim's message. Finally, with $Modulo\ IBN$, the attacker has to guess which $\|IBNSpan\|$ slot to inject its message. Assuming that the attacker only has to guess IBN, Equation 4.5 could be applied to estimate a probabilistic *lower bound* for the *prevention rate*.

Error Passive. The fastest way to push a victim to the *error-passive* state requires at least 16 successive collisions. The *prevention rate* for this scenario is: $P_{prevent} \geq 1 - (1/\|PSpan\|^{16})$.

Bus-Off. The fastest *bus-off attack* requires the attacker to cause 32 successive collisions. The *prevention rate* of this scenario is: $P_{prevent} \geq 1 - (1/\|PSpan\|^{32})$.

4.5.4 Flooding Attacks

ZBCAN prevents flooding by suspending the attacker using the *instant bus-off* technique. Since the attacker's ultimate goal is to cause victims to be unable to access the bus and eventually drop their messages, the success of flooding attacks is measured by the *drop rate* ($rate_{drop}$) they cause to messages. We define our *prevention rate* of flooding attacks to be $rate_{prevent} = 1 - rate_{drop}$. $rate_{prevent}$ will differ from one system to another depending on factors such as the busload and the ID allocation of the network more than the $\|IBNSpan\|$.

4.5.5 Choosing ||PSpan||

Looking at Equations 4.4 through 4.6, we notice that regardless of the value of $\|PSpan\|$, injections will always be detected, given enough cycles. Therefore, the value of $\|PSpan\|$ could be viewed as mainly affecting the *detection speed*, and the *single injection prevention rate*.

The system designer should initially define their security objectives and possible trade-offs. A high *single injection prevention rate* requires a high $\|PSpan\|$. However, ZBCAN provides high detection rates, even for small $\|PSpan\|$ values. For instance, a $\|PSpan\|$ value as low as $16\ b$ will result in a *single injection prevention rate* $\approx 93.75\%$, but a *single injection detection rate* $\approx 99.61\%$ within a single cycle. This is already higher than most of the current IDSs *detection rates* for *flows*. Meanwhile, its *detection rate* of a malicious flow composed of only two messages is $> 99.97\%$ within a single $cycle$. Within $5\ cycles$, both the flow and single injection *detection rates* for the aforementioned scenarios become $\approx 100\%$. Outside injection attacks, the same $\|PSpan\|$ prevents *error passive* and *bus off* attacks at a $\approx 100\%$ rate. To choose a $\|PSpan\|$ value, a compromise between the security of the system and its performance has to be reached. The busier the system, the smaller the $\|PSpan\|$ values it could afford.

4.6 Evaluation

For a thorough analysis, we evaluated ZBCAN's false positive rate, security, performance, and scalability on a testbed using artificial data, on a testbed using ExpVehicle's data, and finally on ExpVehicle's CAN bus.

Trusted Officer Platform. We use a Renesas RA6M5 MCU board as the *officer*. RA6M5 MCU runs on an ARM Cortex M-33. It offers memory and peripheral access isolation, secure boot-loading and processing with TrustZone, a CAN module, and a GPIO module.

Pseudo Random Function. We use Chaskey [76], an open source *PRF* that takes $\leq 0.5\ ms$ to generate a $Seqlength = 128\ b$ on an Arduino Uno board and $\approx 1.9\ \mu s$ on the RA6M5.

Zero-point. As explained in Sec. 4.3.2, we measured the value of T_O on an Arduino Uno and determined it to be $7\ b$. The *zero-point* is $T_O + T_{Suspend} = 15\ b$ after the IFS.

4.6.1 False Positive Test

Propagation Delay. This delay is proportional to the bus length. To assess its impact, we attached the *officer* and a *reference* message generator to a breadboard, an *agent* to another breadboard, connected the two with a cable, and added a $120\ \Omega$ resistance on each board. We set the *agent* to transmit a message immediately after every *reference* message with $IBN = 0\ b$. We set the cable length to $5\ cm$ and measured the average spacing between the *reference* and *agent* messages in nanoseconds. Next, we changed the cable length from $5\ cm$ to $30\ m$ and repeated the measurement. The difference between the spacing at $30\ m$ and at $5\ cm$ was $\approx 340\ ns$. This means that the *round trip* propagation delay was $\approx 11.33\ ns/m$ and that the *one way* delay was $\approx 5.66\ ns/m$. At a $500\ kbps$ baud rate, for the *round trip* delay to exceed our *officer*'s sampling point of 75%, the cable length has to be $\geq 132.39\ m$, which is too long for a typical CAN bus.

Impact of Clock Skew. In a system composed of 20 ECUs, the smallest d_{skew} was $1189\ b$ and the largest $d_{skew} = 1460\ b$. The details of these measurements are explained in Appendix B.2. To assess the impact of clock skew on the false positive rate, we connected the *officer* and a traffic source, sending a *reference message* every $6\ ms$, to a $500\ kbps$ bus. Next, we connected the ECU with the largest d_{skew} and set it to transmit after every *reference message* with an *ascending* IBN between 0 and $3000\ b$. This means sending the first message with $IBN = 0\ b$, the second with $IBN = 1\ b$ and so on until $3000\ b$, then rolling over to $0\ b$ and repeating. Meanwhile, the *officer* monitored the IBN of each message. We ran this test for $30\ min$. Next, we connected the ECU with the smallest d_{skew} and repeated the test. Fig. 4.10 shows how for both ECUs, the false positive rate is 0% before each ECU's d_{skew}, then increases after exceeding it until it reaches 100%.

False Positive Test With Dummy Messages. To assess whether the *dummy message* solution (Sec. 4.4.3) could keep the *false positive* rate of a network with both propagation delay and clock skews at 0%, we connected the *officer* and a traffic generator to one breadboard and the *agents*

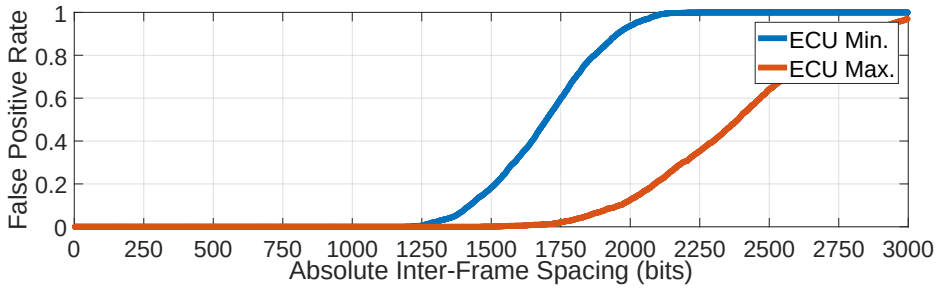

Figure 4.10. Message spacing vs. IBN accuracy.

to another and connected the two with a $30\ m$ cable to maximize the *propagation delay*. We set each *agent* to register and exchange an IBN sequence of an $IBNSpan = 128\ b$ with the *officer* as explained in Sec. 4.3.3. As previously determined (and as explained in Appendix B.2), the system's d_{skew} was $1189\ b$. Nonetheless, since the system's $IBNSpan = 128\ b$ and $1189\ mod\ 128 \neq 0$, we chose $d_{dummy} = 1152\ b$, the biggest value under 1189 whose $mod\ 128 = 0\ b$. Next, we set up the traffic generator to send a dummy message of length $0\ B$ at $1152\ b$ idle time and set up each ECU to send $100K$ messages while following its own IBN sequence. For all ECUs, the *false positive rate* remained 0% after $100K\ messages$.

4.6.2 ZBCAN Security Evaluation on a Testbed

On a $500\text{-}kbps$ CAN bus, We connect the *officer*, 5 ECUs, and a dummy message generator. ECUs are composed of Arduino Uno boards, mcp2515 CAN controllers, and mcp2551 transceivers. One node is used as the attacker. We assume a smart attacker who knows the $IBNSpans$ of the system. Therefore, we provide the attacker with an *agent* similar to the one on all nodes with modifications to launch attacks. Below, we report ZBCAN's evaluation results.

Injection. We set the busload to 34% as in our Impala and evaluated ZBCAN under $\|IBNSpan\|$ values of 16, 32, 64, and $128\ b$ and three types of injection: (1) *Random Injections:* Attacker uses the same PRF with a random seed to try different IDs and IBN values within the $IBNSpan$. (2) *Targeted Injections:* Attacker injects an ID already in use in the network but uses the same PRF to

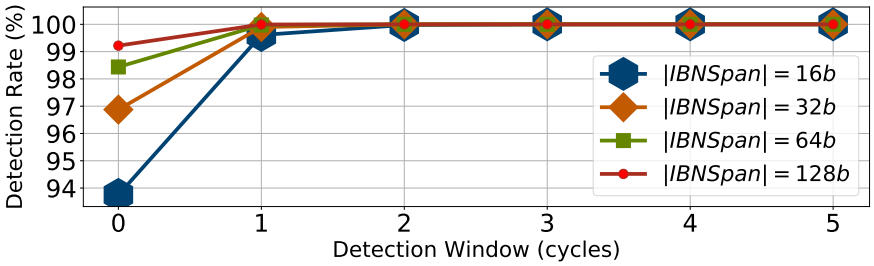

Figure 4.11. Average observed detection rates and windows for targeted injections and replay attacks.

Table 4.2. Observed effectiveness of ZBCAN with different ||IBNSpans|| against different single injection attack types.

Attack	Detection Rate	Prevention Rate Per $\|\|IBNSpan\|\|$			
		16 b	32 b	64 b	128 b
Random Injection	100%	100%	100%	100%	100%
Targeted Injection	100%	93.6%	96.9%	98.5%	99.1%
Replay	100%	93.8%	96.8%	98.4%	99.3%

randomly select IBN values within the $IBNSpan$. (3) *Replay:* Attacker replays a message with the same ID and IBN as the last message on the bus.

As shown in Table 4.2, the *prevention rates* of the *targeted injection* and *replay* attacks were similar to one another and within Equation 4.5's estimated range. The *random injection attack's prevention rate* was 100%. All attacks were detected at a 100% rate. However, Fig. 4.11 shows that the *detection window* for the *targeted injection* and *replay* attacks depended on the $||IBNSpan||$. This confirms our previous analysis in Sec. 4.5.5 and the findings of Equation 4.4, which state that the $||IBNSpan||$ impacts the speed of detection, not whether the attack would be detected.

Error Handling. With $||IBNSpan||$ values of 16, 32, 64, and 128 b, we set the attacker to inject collisions to a victim ECU using *simultaneous transmission*, to count the number of collision attempts, and the number of successful collisions. We set the victim to record the number of times it is pushed to the *error-passive* or *bus-off* states. As shown in Table 4.3, *error-passive* and *bus-off* attacks were prevented at a rate of 100%. *Collision injections* were prevented at rates ranging

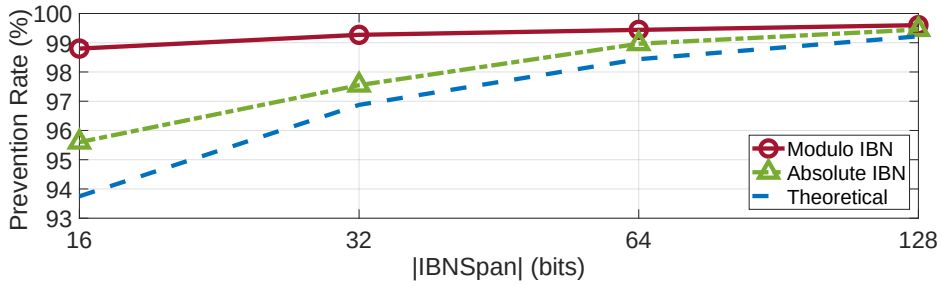

Figure 4.12. Observed prevention rates of the collision injection attack with Modulo IBN turned on and off.

Table 4.3. Observed effectiveness of ZBCAN with different ||IBNSpans|| against different error handling attack types.

Attack	Prevention Rate Per $\|\|IBNSpan\|\|$			
	16 b	32 b	64 b	128 b
Collision Injection	98.8%	99.3%	99.4%	99.6%
Error-Passive	100%	100%	100%	100%
Bus-Off	100%	100%	100%	100%

between of 98.8% and 100%. To evaluate the analysis in Sec. 4.5.3, we disable the $Modulo\ IBN$ feature of our system, rerun the experiment. Fig. 4.12 plots the *prevention rate* with $Modulo\ IBN$ on and off, as well as the theoretical lower bound.

Flooding. We operated the testbed under the following busloads: 10, 20, 30, and 40%. We set one of the nodes as a receiver to confirm message reception. We set the attacker to send back-to-back messages with $ID = 0_h$, and we measured the message *drop rate* with the *officer* disconnected. For all busloads, the *drop rate* was 100%. We repeated the same test with the *officer* connected and we achieved the following *drop rates*: $0\%, 0\%, 0\%$, and 0.67% for the respective busloads.

4.6.3 Performance with Real Vehicle Data

To evaluate the performance impact of ZBCAN if installed on a real vehicle, we emulated the traffic of a test vehicle containing 4 ECUs, 50 IDs, and a 34% busload, whose data is shown in Appendix Table B.1. We applied ZBCAN under three $IBN\ settings$: (A) 3 Priority groups,

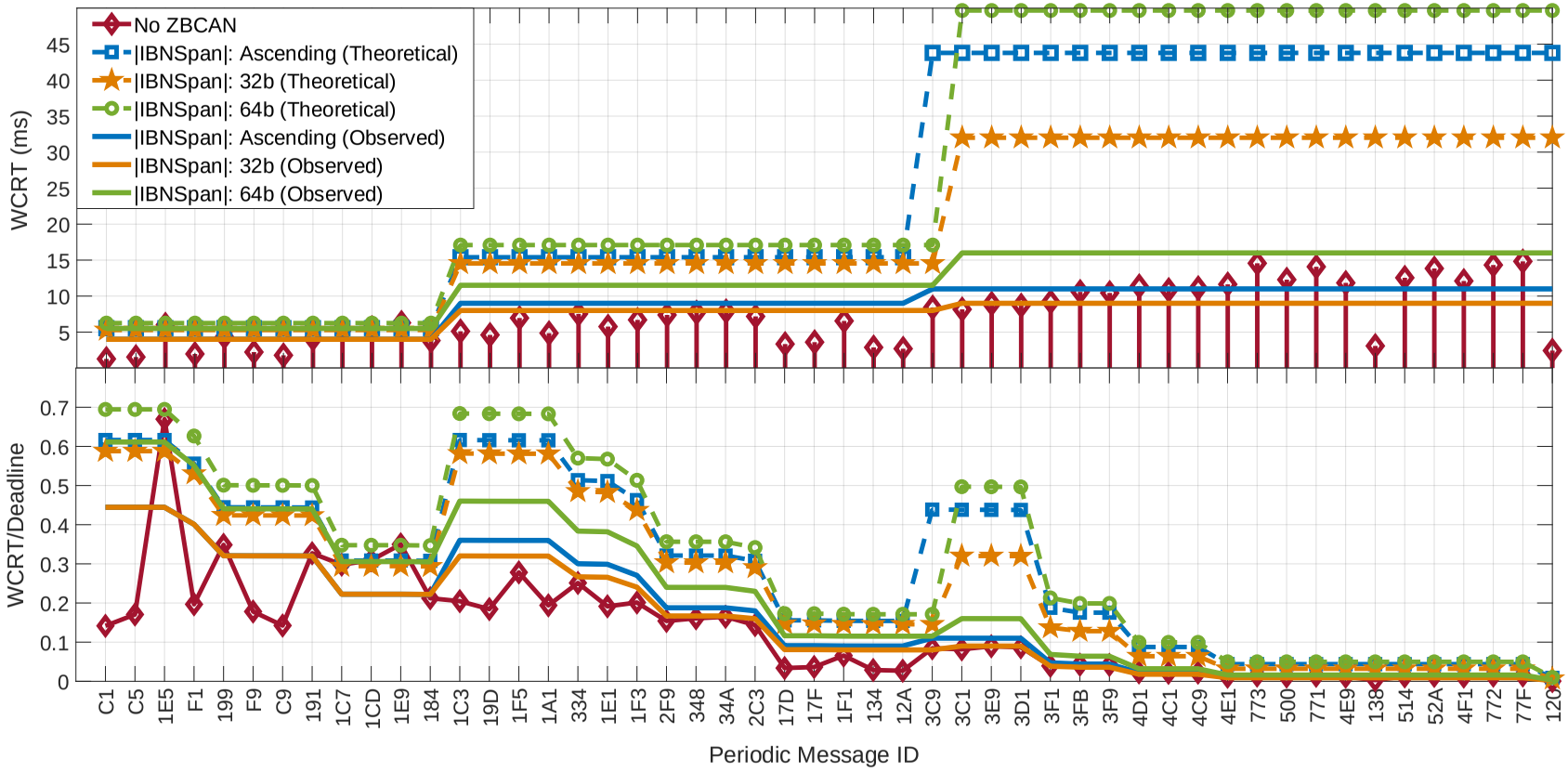

Figure 4.13. WCRTs (absolute and as ratios of message deadlines) for ExpVehicle traffic with and without ZBCAN.

each $\|PSpan\| = 32\ b$. (B) 3 Priority groups, each $\|PSpan\| = 64\ b$. (C) 3 Priority groups with ascending priority spans where $\|PSpan_0\| = 32\ b$, $\|PSpan_1\| = 64\ b$, and $\|PSpan_2\| = 128\ b$.

Grouping. We ran Alg. 4 under the three aforementioned $IBN\ settings$. We assumed that all messages are time-sensitive and that each message's deadline is equal to its period length. Alg. 4 determined $Ratio_{safe}$ to be 0.59, 0.7 and 0.62 for $settings\ A$, B, and C, respectively. The small value of $Ratio_{safe}$ for all settings guarantees that no time-sensitive messages will violate their deadline. Appendix Table B.1 shows the group boundaries for each $setting$.

Observed and Theoretical WCRTs. The top of Fig. 4.13 shows the observed and the theoretical WCRT of each ID for all three $IBN\ settings$ and without ZBCAN. The bottom shows WCRT/Deadline ratio for each ID. The IDs are arranged from left to right based on their period lengths, with the shortest period being on the left. As shown, no message ID violated its timing deadline. The difference between the theoretical WCRT with and without ZBCAN is small on the left, but gets bigger as we move towards IDs with big periods (right). However, when looking at the WCRT as a ratio of each ID's deadline, this increase becomes trivial. For example, while the theoretical WCRT for ID 120_h at $\|IBNSpan\| = 64\ b$ approaches $50\ ms$, it has a period of $5\ s$, making the theoretical $WCRT/Deadline \leq 1\%$.

Without ZBCAN, the maximum $WCRT/Deadline$ ratio lies at $ID : 1E5_h$ around 0.7, the same as with ZBCAN, while the observed ratio is much smaller. Note that the theoretical WCRT is always greater than or equal to the observed WCRT, and that the difference gets bigger as we move right. This is because our WCRT analysis in Sec. 4.4.1 is too pessimistic. However, even this pessimistic theoretical WCRT is still below the timing deadline of all messages.

Observed Average and Minimum Response Times. For all $IBN\ settings$, the average delay for P_0 was $\approx 500\ \mu s$, and $< 1\ ms$ in 90% of the cases. For P_1 and P_2, the average delays were $\leq 2\ ms$, and $\leq 3\ ms$, respectively. For all groups, $setting\ (B)$ had the largest minimum, average, and maximum response times. Per Equation 4.3, where the $\|PSpan\|$ which influences WCRT the most is $\|PSpan_0\|$. Since $setting\ B$ has the largest $\|PSpan_0\|$, it also has the largest delays.

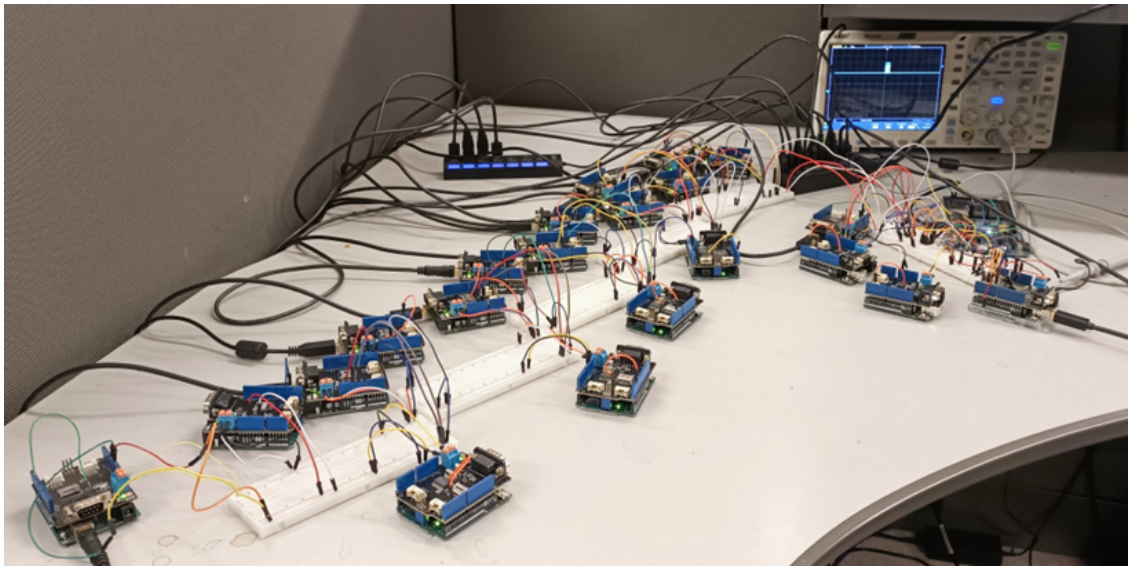

Figure 4.14. Testbed with 20 ECUs (agents) and the officer.

Dummy Messages. d_{dummy}, or the maximum inter-frame space allowed before clock skews start causing inaccuracies was determined to be $> 2\ ms$ (Appendix B.2). Since the average spacing between messages was $\approx 1.6\ ms$, the dummy message did not need to be inserted most of the time, resulting in a busload increase of $< 1\%$.

4.6.4 ZBCAN Scalability Evaluation

To evaluate ZBCAN's performance on a system with a large number of ECUs and safety critical messages, we setup a testbed with 20 ECUs and 100 message IDs. All messages were assumed to be safety critical and time sensitive, with their timing deadlines shown in Appendix Table B.2. Note that messages $1, 6$ and F are aperiodic. Fig. 4.14 shows our scalability testbed.

Grouping. We ran Alg. 4 under a 3-priority $setting$, a message-length$= 8\ B$, a $\|PSpan\| = 64\ b$, and an $\|IBNSpan\| = 192\ b$. The calculated theoretical WCRTs were $7.3, 37.1, 96\ ms$ and the $Ratio_{safe}$ to be $0.73, 0.74$ and 0.38 for groups $0, 1$ and 2. For all settings, we evaluated the performance aspects below. Appendix Table B.2 shows the group assignments.

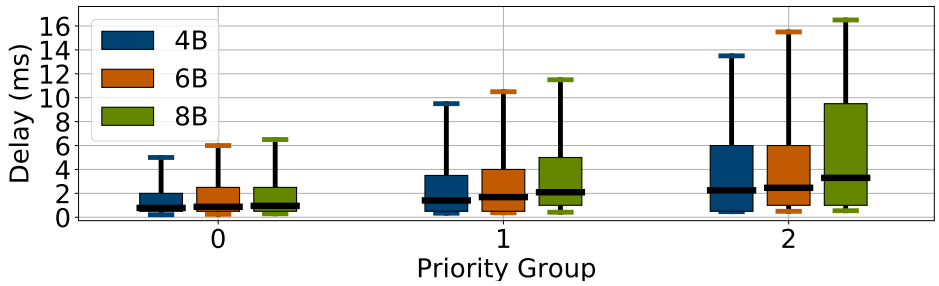

Figure 4.15. Observed WCRTs vs. message length

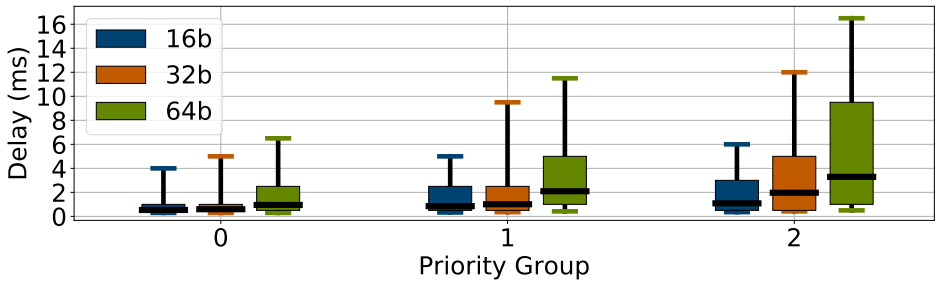

Figure 4.16. Observed WCRTs vs. ||PSpan||.

Observed WCRTs. We set the transmission rates of the aperiodic messages to the highest to achieve the worst possible delays on the bus. The observed WCRTs were $\approx 6.5, 11.5$ and $17\ ms$ for priority groups $0, 1$ and 2, respectively. No messages violated their deadlines or their theoretically calculated WCRTs as shown in Appendix Table B.2, which proves the accuracy of our time model and that ZBCAN is safe to use in time-sensitive networks.

WCRT and Message Length. To evaluate the impact of message length on WCRT, we repeated the evaluation with message lengths set to 4 and $6\ B$ and observed the WCRTs. Fig. 4.15 shows the average, maximum, minimum, 10^{th} and 90^{th} percentile observed WCRTs. As shown, increasing the length clearly increases the delay in every *priority group*.

WCRT and $\|PSpan\|$. To evaluate the impact of $\|PSpan\|$ on WCRT, we reset the message lengths to $8\ B$ and, using the same group assignments, repeated the evaluation with $\|PSpan\|$ of all groups

set to 16 b then 32 b and observed the WCRTs. As shown in Fig. 4.16, increasing $\|PSpan\|$ clearly increases the delay of messages in every *priority group*.

WCRT and Number of ECUs. To assess the impact of the number of ECUs on WCRT, we reset all lengths to 8 B, all $\|PSpan\|$s to 64 b and repeated the evaluation with 10 ECUs instead of 20 (ECU sends 10 IDs instead of 5). As shown in Fig. 4.17, the impact of increasing the number of ECUs on the observed delay does not show a clear up or down trend. For instance, while the observed WCRT for group 1 slightly increased, it slightly decreased for group 2. Meanwhile, the average WCRTs for all groups remained almost the same.

Per Sec. 4.4.1, the number of ECUs is not expected to directly impact the WCRTs. Similarly, per Alg. 4, the theoretical WCRTs are the same. Nonetheless, the WCRTs are not identical. These differences are mainly due to the *queuing jitter* that changed as a result of changing the number of IDs per ECU. Specifically, while the maximum jitter in both scenarios was $\approx 750\ \mu s$, the average jitter was slightly higher for the 10-ECU scenario, resulting in a very slightly higher average WCRT for the 10-ECU scenario. The jitter's standard deviation, however, was higher for the 20-ECU scenario, resulting in a slightly higher maximum WCRT for priority group 1.

Memory Consumption. We equipped a busy ECU transmitting 20 IDs with a ZBCAN *agent*. The memory consumption of all the ECU's and *agent*'s variables was $\approx 1.72\ kB$.

Sequence Extension Overhead. With $L_{seq} = 128\ b$ and a $\|PSpan\| = 64\ b$, a busy *agent* with 20 IDs, each with a period of 25 ms, performed a *sequence extension* operation every $\approx 26.25\ ms$ (P_{ext}). Each operation took $\approx 0.5\ ms$ (d_{ext}). The *agent's* $O_{ext} = d_{ext}/P_{ext}$ ratio was $0.5 * 100/26.25 \approx 1.91\%$. On the *officer's* side, each extension operation took $\leq 1.9\ \mu s$. On a busy system with a message observed every $0.5\ ms$, a *sequence extension* operation happened every $\approx 10.5\ ms$, resulting in a an O_{ext} of $0.0019 * 100/10.5 \approx 0.018\%$.

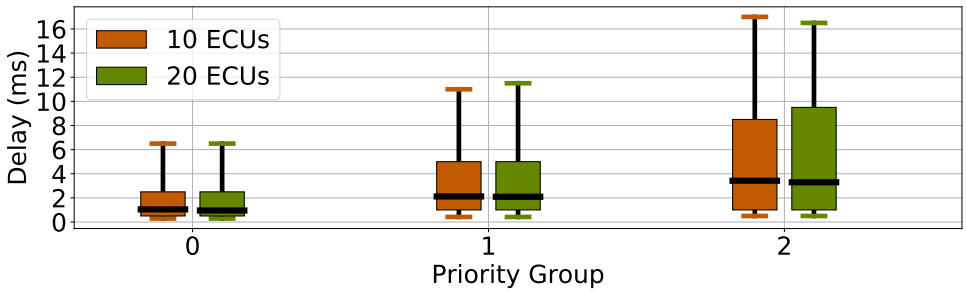

Figure 4.17. Observed WCRTs vs. numbers of ECUs.

4.6.5 ZBCAN on a Real Vehicle

Incremental Deployment. Incrementally adding a protected ECU to an unprotected bus is different from adding it to a protected one. Specifically, only the protected ECU's messages will have IBN delays. Therefore, we expect such an ECU to have higher delays than one deployed on an already protected bus (Sec. 4.6.3). To evaluate these delays, we attached the *officer* and an ECU equipped with ZBCAN to the CAN bus of a real vehicle. We set the *agent* to transmit at a period $= 50\ ms$ and under the following $\|IBNSpans\|$: $16, 32, 64$, and $128\ b$. Fig. 4.18 shows the average, minimum, maximum, 10^{th}, and 90^{th} percentile response times.

Table 4.4. Effectiveness of ZBCAN against flooding attacks.

Setting	Testbed				ExpVehicle
	10%	20%	30%	40%	
Prevention Rate	100%	100%	100%	99.33%	100%

Flooding. We attached an attacker to send back-to-back messages of ID$= 0_h$ with the *officer* disconnected. We connected an additional receiver node to confirm the reception of messages. The message drop rate was 100%, which means no messages whatsoever were being received. We repeated the experiment with the *officer* connected and the drop rate becomes 0%. This means

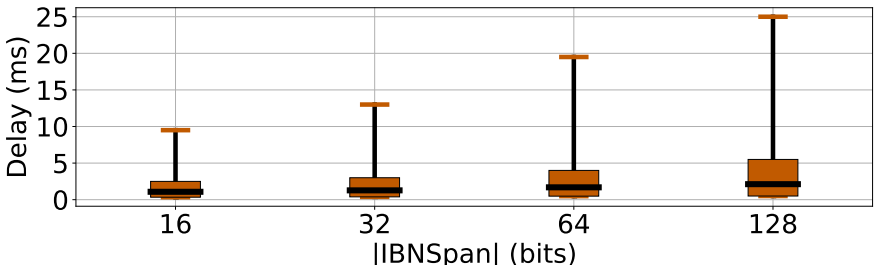

Figure 4.18. Observed WCRTS on ExpVehicle.

that the attack *prevention rate* with the *officer* connected was *100%*. Table 4.4 shows ZBCAN's prevention rates of flooding in different testing conditions.

4.7 Benchmark Comparison

Since ZBCAN is versatile, it is difficult to compare its effectiveness against its entire set of attacks with systems that may not defend against the same set. Instead, we make three separate comparisons between ZBCAN and other systems.

Compared to IDSs. While most defense systems focus only on *injection attacks*, ZBCAN protects against a wider attack-set, encompassing *injection*, *error handling*, and *flooding*. Further, while most intrusion detection systems are able to detect attacks composed of message flows with a high degree of accuracy, ZBCAN offers security guarantees to single messages as well as flows. These abilities guarantee that, unlike other IDSs, gradual, intermittent, and single-message attacks will not pass unnoticed. They also allow ZBCAN to extend some of its detection abilities to prevent attacks. Table 4.5 compares ZBCAN's evaluation results with the evaluation results of other IDSs against the same attack set. As shown, ZBCAN's *prevention rate* compares with the best IDS's *detection rates*. Meanwhile, its *detection rate* is 100%.

Table 4.5. How ZBCAN's evaluation results at $\|\text{IBNSpan}\| = 64$ b compare with other intrusion detection systems.

IDS		Targeted Injection	Replay
Scission[27]		96.8%-98.5%	96.8%-98.5%
Clock-Skew[24]		97%	97%
ZBCAN	Detection	100%	100%
	Prevention	98.5%	98.4%

In conclusion, ZBCAN protects against a bigger attack set than any other defense system, does not require dynamic retraining, offers higher detection abilities of attack flows as well as single attack instances than any other IDS, and finally, attack prevention as well as detection.

Table 4.6. Comparing the probability of a single injection going undetected with different benchmarks.

Defense	P_{undet}			Message Bytes	Busload
	1 cycle	5 cycles	10 cycles		
Leia[48]	$1/2^{64}$	$1/2^{64}$	$1/2^{64}$	2	+100%
LCAP[45]	$1/2^{16}$	$1/2^{16}$	$1/2^{16}$	2	+0%
CACAN[77]	$1/2^{8}$	$1/2^{8}$	$1/2^{8}$	1	+100%
IA-CAN[49]	$1/2^{32}$-$1/2^{8}$	$1/2^{32}$-$1/2^{8}$	$1/2^{32}$-$1/2^{8}$	1-4	+0%
ZBCAN	$1/2^{14}$-$1/2^{8}$	$1/2^{42}$-$1/2^{24}$	$1/2^{77}$-$1/2^{44}$	0	+0-1%

Compared to Cryptographic Solutions. We define the metric P_{undet}, which quantifies the probability of a single injection going undetected. As shown in Table 4.6, while P_{undet} within $one\ cycle$ is lower for Leia and LCAP, it is constant with w. As explained by Equation 4.4, this probability is 0 with ZBCAN, given enough cycles (w). As shown, at $w = 10\ cycles$, ZBCAN's P_{undet} is the lowest. Further, these systems use message bytes and some of them double the busload. ZBCAN is the only one to use $zero$ message bytes and cause no or a very small busload increase. Finally, while P_{undet} for IA-CAN and CACAN are comparable to ZBCAN's at $w = 1\ cycle$, an attacker

may exhaustively try injecting all different combinations. With ZBCAN, this cannot happen since the *officer* will suspend nodes that try this approach.

Table 4.7. CANARY and ZBCAN are effective defenses against error handling and flooding attacks.

Defense System	Response Time	Attacker Isolation	Hardware Changes
CANARY[64]	5ms-100ms	Partial	1 Guardian Node + 8 Relays + Wiring
ZBCAN	22-72 us	Full	1 Officer Node

Compared to Other Solutions. CANARY is one of the few defense systems that addressed *error handling and flooding attacks*. As shown in Table 4.7, in addition to the expensive costs of wiring and adding relays, relays work by isolating entire sections of the CAN bus, which may result in the isolation of benign nodes, together with the attacker. Moreover, relays often have high relaying times, resulting in attacks taking place for a long time before soliciting a response.

4.8 Discussion

Intrusion Confinement. ZBCAN provides *intrusion confinement* in two ways. First, since *agents* do not share the same keys or sequences, a compromised ECU *agent* cannot predict the IBN sequences of other nodes and hence cannot inject messages impersonating other nodes or reliably inject collisions. Second, a compromised node cannot launch *flooding or error handling* attacks against other nodes, since the *officer* will actively interrupt any such attempt.

Time Triggered CAN. TTCAN systems use *matrix scheduling*, in which each message is expected to be transmitted during a specific interval. No other message is allowed to be sent during this interval. Additionally, TTCAN systems use *reference messages* to provide synchronization between nodes. These two factors make TTCAN systems a good fit for ZBCAN. Namely, *reference messages* eliminate the need for additional *dummy* messages. Similarly, prescheduling messages eliminates

the need for *priority groups*, increases the security of the system, eliminates the *interference delay* in Equation 4.3, and significantly improves the WCRT. The only minor change these systems may need is to increase the acceptance window for each message by a distance $= \|IBNSpan\|$.

ZBCAN Controller. CAN controllers have the hardware required to monitor and change message spacing (e.g., *suspend transmission period*). With slight modifications, the functionality of the *agents* could be implemented in the form of controllers, eliminating the overhead on the ECU's side.

Content Authentication. Since we are trying to make ZBCAN as lightweight as possible, we only discussed transmitter authentication. However, ZBCAN could be easily extended to include content authentication by calculating a hash or MAC for each message and then XORing the result with IBN_{sc}. On the *officer's* side, the *officer* will have to read the entire payload as opposed to reading only ID bits, calculate the keyed hash or MAC, then XOR it with the expected IBN_{sc} value to see if the result is equal to the measured IBN.

Other Possible Extensions. In addition to *content authentication*, the *officer*'s design could be easily extended to support more sophisticated operations. For example, it could be extended to provide protection to several buses operating at varying levels of security (e.g., confidential, secret, top secret, etc.) similar to monitoring solutions that have been designed for other communication buses[78]. It could also be extended to offer deep packet inspection or content authentication by checking the DLC, RTR, or payload fields. For example, it could be provided with the used payload fields, signal boundaries, and accepted value ranges specific to each message ID to provide deep packet inspection as has been proposed on other buses [79, 80].

4.9 Limitations

Officer Failure. Similar to most IDSs, the *officer's* failure removes the security it provides. However, the *officer* is not a filter, a bottleneck, or a gateway that messages go through before reaching the bus, rather, it is connected in parallel as any other ECU. We designed ZBCAN this way so that even if it fails, it *fails safe*. Further, applying IBN values, transmission, and reception are the *agent's*, not

the *officer*'s, responsibility. Consequently, if the *officer* fails, *agents* continue following their IBN sequences normally. This means that while the protection against injection and flooding attacks will be removed, it remains intact against *error handling* attacks, since they do not rely on the *officer*, but the unguessability of the IBN that the attacker needs for the collision.

Compromised Officer. The *officer* is assumed to be trusted. This assumption is not unique to ZBCAN but common to all IDSs. Although existing IDSs only monitor the network, they still have access to the bus through a CAN controller and/or special pins. Both channels could launch attacks if the IDS is compromised. Nonetheless, to minimize the probability of such a scenario, we used a board that provides *secure boot-loading*, *secure processing*, *secure memory access isolation*, and *peripheral access isolation*. We could use these features in several ways to elevate the security posture of the *officer* and minimize the likelihood of its compromise.

For example, we could use the secure boot-loading process for the officer node to ensure that only authenticated and authorized software and firmware updates are loaded during the boot sequence. This way, we protect against unauthorized modifications to the *officer*, such as firmware tampering or unauthorized updates. We could also use the *secure memory access isolation* feature to store critical information, such as the keys and IBN sequences in isolated memory regions, and ensure that they remain protected and inaccessible to unauthorized parties. The secure processing capability could handle the key management and the IBN sequence extension. By utilizing the secure execution environment, the *officer* node can perform cryptographic operations, such as key generation, key derivation, and sequence extension within a trusted space. To protect against the risk of an unauthorized malicious application taking over the GPIO or CAN peripherals, we can use the *peripheral access isolation* feature and restrict access to these peripherals to trusted applications.

Most importantly, we should stringently restrict physical access to the officer node to authorized personnel only to prevent unauthorized tampering or access to the *officer* node. Finally, except for the channels connecting the *officer* to the bus, we could disconnect any other connections to or from the *officer*. This *air-gapped-like* setting greatly elevates its security posture.

In-Group Priority Inversions. Although ZBCAN guarantees that a message belonging to *priority group* P_n will always have a higher priority than one belonging to group P_{n+1}, it does not guarantee

the priority hierarchy within the same group. Nonetheless, ZBCAN guarantees that even if an in-group priority inversion takes place, every message in the group is guaranteed to be *schedulable* and to arrive at its destination without violating any timing deadlines.

Unschedulable Systems. We assume all messages are time sensitive and use Alg. 4 to plan the system's schedulability. However, it is theoretically possible to find unschedulable systems. In such cases, we have to find a trade-off. This could involve the decision of changing, lowering, or even dropping, the security of certain messages, to guarantee the schedulability of all messages. For instance, assuming a 3-priority system with each $\|Pspan\| = 64\ b$, in Appendix Table B.2, if we lower the periods/deadlines of the first five messages to $5\ ms$, the system becomes unschedulable unless we lower the span of all groups to $32\ b$. Further lowering the periods/deadlines of the first five messages to $3\ ms$, renders the system unschedulable again. In this case, we may drop the security of the first 5 messages by adding them to group 0, then setting $\|Pspan_0\| = 0\ b$, while keeping the protection for other groups to preserve the system's schedulability.

Corrupt Payloads. The discussed design of ZBCAN does not prevent an ECU from corrupting the payload of its own messages. However, the *officer* could be easily extended to check the DLC, RTR, and payload fields to detect or prevent this scenario, as mentioned in the discussion section.

4.10 Conclusions

We proposed ZBCAN, a novel defense system that exploits inter-frame spacing to protect against the most common CAN attack vectors. ZBCAN offers *attack detection, prevention, individual message guarantees, intrusion confinement, incremental deployability*, and full *backward compatibility* without using any message fields. It also avoids making heavy use of computationally expensive operations such as encryption. We introduced a novel and instant way to suspend nodes called *the instant bus-off* technique, which we used for defense purposes against intruding nodes. We proved the schedulability of messages on systems implementing ZBCAN by offering a theoretical worst-case response time analysis for such systems. We offered a probabilistic security analysis for ZBCAN against different attack types. Finally, we proved the applicability of our system by

evaluating different aspects of it on a testbed using artificial data, then on a testbed using a real vehicle's data, and finally on a real vehicle's CAN bus.

5. CONCLUSIONS AND FUTURE DIRECTIONS

The transition from purely mechanical systems to computerized cyber-physical systems has been incremental and often unpremeditated, resulting in systems with limited security measures. With cyber-physical systems, breaches often begin in the cyber world, but their impact manifest in the physical world. In many cases, this causes life-or-death scenarios. Correspondingly, the importance of addressing the security vulnerabilities in today's cyber-physical systems cannot be overstated. Unfortunately, the existing security approaches are often reactive rather than proactive. We proposed a proactive vulnerability identification and defense construction approach for cyber-physical systems, with CAN serving as a relevant case study. This approach involves systematic vulnerability scanning, root cause investigation, and the construction of defense systems that consider the nature of the system and protect against multiple vulnerabilities simultaneously. The ultimate aim of this approach is to enhance the security of cyber-physical systems and mitigate potential risks. We emphasize that the findings and techniques presented here do not apply only to CAN but also to other cyber-physical systems that share similar performance and security challenges.

We began this dissertation with a vulnerability identification phase. Specifically, following our identification of a novel and alarming attack type against CAN, which targets its error handling and fault confinement mechanism, we decided to formally investigate the security posture of this fundamental and previously overlooked mechanism of the CAN standard. We built an automated vulnerability identification tool, named the *CAN Operation eXplorer (*CANOX*)* [38], which systematically scans CAN's error handling and fault confinement mechanism for vulnerabilities. It explores the impacts of operating outside the default *error active* state to identify possible vulnerabilities in the CAN standard as such impacts have never been studied in the literature. CANOX places a CAN node in a controlled environment, sets its operation and error state, systematically changes the operational conditions of the node and its environment, and monitors certain behavioral metrics to identify the conditions that result in unexpected node behaviors. CANOX uses this dynamic approach to avoid the problems associated with formal model checking such as inaccurate modeling.

Using CANOX, we identified three major undiscovered vulnerabilities in the CAN standard [39]. These vulnerabilities allow an attacker to shut down an ECU by attacking a single message, persistently prevent an ECU from recovering from a shutdown, or perform black-box network

mapping. To show the gravity of the discovered vulnerabilities, we constructed a single, end-to-end, multi-staged attack that exploits all of the discovered vulnerabilities. In it, an attacker with no previous knowledge of the vehicles internals maps the vehicles CAN bus, identifies a safety-critical ECU, swiftly silences it, and persistently prevents it from recovering. we validated the applicability of the attack by evaluating it on a testbed and a real vehicle.

Due to the impact of the discovered vulnerabilities on CAN systems, we first provided some mitigation approaches as a temporary solution. Next, we reported the three vulnerabilities to the Robert Bosch Product Security Incident Response Team (PSIRT), to the Cyber-security and Infrastructure Security Agency (CISA) through the CERT-VINCE portal, and to the International Organization for Standardization (ISO). ISO referred us to the American National Standards Institute (ANSI), which directed us to the Society of Automotive Engineers (SAE). SAE acknowledged our contributions and submitted the vulnerabilities to a committee for review and consideration in the next revision. Although our USENIX paper was accepted in January [38], we requested that it be placed under embargo until August, to give a chance to all the concerned parties to deploy their own security measures. Finally, per their request, we gave a talk to the Automotive Information Sharing and Analysis Center (AutoISAC) to explain the vulnerabilities and the attacks that could exploit them in detail. This proactive engagement with industry stakeholders and standardization bodies underscores our commitment to driving impactful change and fostering the adoption of robust security measures within modern cyber-physical systems.

We then transitioned into the defense construction phase. We began this phase by surveying the existing CAN defense systems and approaches. Unfortunately, we noticed that although several CAN defense approaches have been proposed, none of them could be widely adopted for reasons inherent in their design, such as their unaffordable overhead, increased busload, dropped data rate, lack of single-message detection, lack of attack prevention, or inaction against non-injection-based attacks. To address these shortcomings, we designed Zero-Byte CAN (ZBCAN) [81], a low-overhead defense system that offers intrusion detection as well as prevention, and single-message detection, without increasing the busload, dropping the data rate, or using computationally expensive operations such as encryption. Instead, it leverages message interarrival times alone to protect against the most common CAN attacks, including injection, masquerading, impersonation, fuzzing, flooding, collision injection, voltage corruption, and bus-off attacks.

ZBCAN consists of a trusted monitor node, able to stop messages during transmission, called the *officer*, and a set of software *agents*, installed on every ECU. Each ECU privately agrees on a secret, non-repeating, and unique sequence of inter-frame spaces, called the *In BetweeNs* ($IBNs$), with the *officer*, and then applies these sequences upon outgoing messages. If the *officer* detects a with the wrong IBN or ID, it stops it precisely after the ID portion, thus invalidating it and preventing it from being received by any ECU. Depending on the *officer's* setting, it may ignore the message, issue a warning, stop it, or disable its transmitter. This way, we could prevent several attacks at once. Namely, *bus-off, network mapping, or voltage corruption* attacks rely on a technique called *simultaneous transmission*, where attackers have to transmit a message exactly at the same time as their victim. With ZBCAN, they need to guess every message's IBN value to transmit simultaneously. Similarly, *injection* (including masquerading, impersonation, etc.) and *flooding* attackers need to guess the correct IBN value for every message, or else they will be stopped by the *officer*. Aside from attaching the *officer* to the bus, ZBCAN does not require any hardware changes. Further, since it uses no message fields, it could be combined with any solutions that use them.

For inclusivity, we evaluated several security, performance, and scalability aspects of ZBCAN on a testbed and an actual vehicle. With a 0% false positive rate, ZBCAN achieved a detection rate of 100% for injection, masquerade, and replay attacks. Relative to its prevention abilities, ZBCAN achieved prevention rates of 100%, 100%, 99.4%, 99.33%, and 98.5% for error-passive, bus-off, collision injection, flooding, and injection attacks, respectively.

In addition to protecting against the most common CAN attacks, detecting attacks at a 100% rate, and offering attack prevention, ZBCAN is backward compatible and cost-effective, allowing for its wide-scale adoption. If implemented at the controller level, a project we are currently discussing with our industry partners, ZBCAN agents would have zero performance overhead. Most importantly, ZBCAN uses a channel present in all serial buses, message interarrival times. Considering this channel for security purposes could protect all serial buses, not only CAN. We plan to file a patent that includes my design recommendations for utilizing interarrival times for security. We are also in the process of preparing a technology transfer of ZBCAN to several technology firms.

In conclusion, our research highlights the importance of proactive security measures in cyber-physical systems. By systematically identifying vulnerabilities and constructing defense systems, we can enhance the security of cyber-physical systems and mitigate potential risks. As we continue

our research, we aim to further bolster the security posture of modern vehicles and other critical cyber-physical systems, promoting their safe and reliable operation.

5.1 Future Directions

In this dissertation, we focused primarily on the CAN bus. However, many other networked, cyber-physical, and autonomous systems could utilize the proposed approach. Specifically, starting with a vulnerability identification process before proceeding to find mitigation approaches invariably improves the security and reliability of these technologies. Below, we discuss three fast-developing areas that could benefit from our proactive approach by developing vulnerability identification frameworks, security protocols, policies, and procedures, as well as intrusion detection systems: emerging vehicular networks, connected vehicles, and autonomous vehicles.

Emerging Vehicular Networks. For decades, CAN has been the dominant in-vehicle communication standard. Change, nonetheless, is the only constant. With the ever-expanding computerization process of vehicle functions, a need for faster and more time-sensitive standards has emerged. Consequently, Non-CAN-based technologies (e.g., FlexRay) have come out [82]. Other CAN-based standards such as CAN-FD and TTCAN have appeared in response [83, 84]. With the recent advent of autonomous vehicles, data-prolific devices such as cameras and lidars have become increasingly present. Accordingly, high-bandwidth technologies (e.g., Ethernet, SerDes, etc.) have begun finding their way onto vehicular platforms [85–88].

Despite the expanding use of these technologies, CAN-based networks have remained the backbone of most vehicles due to their robustness, non-destructive arbitration, lack of need for a switch, and minimalistic wiring requirements. Weighing $50 - 70kg$, the wiring harness is already one of the most expensive parts of the vehicle [89] and is currently the third heaviest part after the engine and the chassis [90]. Provocatively, an emerging suite of *vehicular ethernet* standards, supporting high-bandwidth communications while requiring less wiring is now challenging the status quo. One variant (10BASE-T1S) even supports a single pair of wires and a bus topology [91], eliminating the need for a switch. To remain competitive, the CAN community is developing CAN XL [92], providing higher bandwidth while retaining low wiring costs.

Regardless of how this technological race unfolds, studying the security aspect of these emerging standards is crucial. Our suggestion is to start by designing vulnerability scanning tools for these technologies; whether they should take a dynamic approach, similar to CANOX [38], or a more formal one, such as static analysis or model checking, depends on the nature of each technology. After identifying the potential security problems associated with each technology, we should intend to devise defense strategies that protect against a wide array of threats, similar to ZBCAN [81], and could be used across different implementations while at the same time considering their performance impact. Finally, our aim should be to investigate the effectiveness of these solutions by running experiments on physical vehicles. Overall, the ultimate goal is to better understand these emerging technologies and to create defensive approaches that elevate their security posture.

Connected Vehicles. Connected vehicle technologies (V2X) aim to provide vehicles with more external connectivity. Their ultimate goal is to provide more safety and comfort for the passengers and improve road conditions. V2X technologies could revolutionize the way we view and experience transportation. Nonetheless, they also face many security issues as they have been proven vulnerable to various threats [93, 94]. These threats could result in the interception or manipulation of data, leading to serious safety concerns for passengers. Such threats have raised concerns about the security of V2X, as it is imperative that they remain protected. Pertinent to the Security Credential Management System (SCMS) [95], various weaknesses have been identified. The most central among which is that it is chiefly intended to guarantee the transmitter's identity, not what they transmit. This means that a malicious actor could easily manipulate the data and messages sent by their vehicle to cause problems to other vehicles and transportation systems.

To secure these technologies, we suggest starting by first identifying any potential security vulnerabilities and analyzing the capabilities of various threat actors. Ultimately, the plan is to develop defense strategies and systems to defend against potential attacks on V2X communications. Since most of these protocols are not yet widely deployed, the goal is for these strategies and recommendations to be incorporated into the protocols and mitigate their security risks before their wide-scale deployment. This could include developing a risk assessment framework for these technologies, detailing the techniques and methods used by each threat actor, and outlining counter-measures that could protect against any compromise attempts. Overall, the aim is to proactively

analyze the security vulnerabilities of existing and emerging CV technologies, identify potential threats, and develop defense strategies for them to make sure these protocols remain safe and secure. This way, we ensure that these technologies benefit us without compromising our safety or privacy.

Autonomous Vehicles. In order to eliminate the need for human control of vehicles, autonomous vehicles (AVs) must be equipped with sophisticated sensors and machine-learning heavy control components (e.g., perception, mission planning, detection, tracking, etc.). Given the increasing complexity of autonomous vehicles, they have an expansive attack surface. In addition to the attack surfaces present in conventional and connected vehicles (e.g., CAN bus and V2X protocols), the sizeable number of sensors on autonomous vehicles and their control components represent an added surface [96–101]; attackers may use adversarial machine learning and transduction attacks [102, 103] to target the control components and sensors in autonomous vehicles, respectively.

We could begin by splitting the spacious attack surface of AVs into smaller and more manageable surfaces (e.g., sensors, detection module, etc.). This way, each attack surface is better identified. At this point, we could start applying the proactive vulnerability identification and defense construction approach to individual attack surfaces before applying it to the whole system. We expect this process to be challenging for machine-learning-heavy control components, such as detection, since it is not always straightforward to know why an input to a machine-learning algorithm has resulted in a specific outcome. For instance, we may not know why an object was detected as a pedestrian when it was a vehicle, making slightly altered inputs one of the attackers' favorite attack vectors. While a generic solution to this problem is mechanisms such as adversarial training, blindly applying such a solution does not often achieve good results. Often, identifying the vulnerability leading to such behavior could inspire completely different solutions whose results are much better. Even in the case of adversarial training, guided training, using the knowledge obtained during the vulnerability identification phase, could be more effective. In conclusion, a proactive vulnerability identification and defense construction approach is an effective way to secure autonomous driving systems before deployment, as the cost of dealing with breaches in retrospect is punitive.

Securing OTA Updates. The transformation of vehicles into sophisticated computers on wheels has given rise to the concept of Software-Defined Vehicles (SDVs), where the focus shifts from viewing digital components as separate entities cohabiting in the same physical system to treating the entire

vehicle as a single, centralized digital system with several interconnected sub-components. To keep SDVs up-to-date, the industry is increasingly turning towards Over-The-Air (OTA) updates. These updates, similar to those on smartphones and computers, provide a means to continuously enhance SDVs' functionality. OTA updates are typically received first by the telematics unit, which acts as the central gateway for communication within the vehicle. The telematics unit then facilitates the internal distribution of the update to other Electronic Control Units (ECUs).

Although this approach simplifies the update process, the complexity of SDVs and the distributed nature of their ECUs introduce structural vulnerabilities and compatibility issues, turning SDVs into fertile soil for various security threats. For example, a recent OTA update incident unintentionally caused a recall of 40,000+ vehicles. The OTA firmware release, intended to update the calibration values of the electronic power assist steering system, inadvertently resulted in a loss of power steering ability after encountering potholes or bumps [104]. This incident highlights the necessity for robust OTA update mechanisms that thoroughly test for compatibility issues and vulnerabilities.

Our first research goal is to identify and rectify the structural and architectural vulnerabilities inherent in today's SDVs. We should first comprehensively analyze the communication networks, interfaces, and components within SDVs to uncover potential security flaws. Once we gain insights into the vulnerabilities, the next priority is to take a secure-by-design approach and design a robust and secure OTA update system that ensures the security and compatibility of the distributed components within the SDVs. To achieve this, automotive manufacturers must implement their emulation platforms, enabling them to assess the security impact and compatibility of updates before deployment. By adopting such an approach, manufacturers can proactively identify and mitigate potential risks and vulnerabilities before the updates reach the vehicles.

In conclusion, ensuring the security of OTA updates involves two fundamental objectives: the systematic identification of vulnerabilities in today's SDVs and the design of a reliable and secure SDV architecture that avoids those vulnerabilities and includes a carefully planned update system. This way, we can establish a secure foundation that empowers vehicles to embrace technological advancements while prioritizing safety and security on the road.

REFERENCES

[1] Semiconductor History Museum of Japan, *Trends in the semiconductor industry*, https://www.shmj.or.jp/english/trends/trd70s.html, 1970s.

[2] C. in Automation, *Can history*, https://www.can-cia.org/can-knowledge/can/can-history/.

[3] R. Bosch, *CAN specification - Version 2.0*, 1991.

[4] S. Checkoway, D. Mccoy, B. Kantor, *et al.*, "Comprehensive experimental analyses of automotive attack surfaces," in *USENIX Security Symposium*, 2011, pp. 77–92.

[5] C. Miller and C. Valasek, "Remote exploitation of an unaltered passenger vehicle," *Black Hat USA*, vol. 2015, p. 91, 2015.

[6] S. Nie, L. Liu, and Y. Du, "Free-fall: Hacking Tesla from wireless to CAN bus," *Briefing, Black Hat USA*, 2017.

[7] S. Nie, L. Liu, Y. Du, and W. Zhang, "Over-the-air: How we remotely compromised the gateway, BCM, and autopilot ECUs of Tesla cars," *Briefing, Black Hat USA*, 2018.

[8] K. Koscher, A. Czeskis, F. Roesner, *et al.*, "Experimental security analysis of a modern automobile," in *IEEE Symposium on Security and Privacy (S&P)*, 2010, pp. 447–462.

[9] I. Foster, A. Prudhomme, K. Koscher, and S. Savage, "Fast and vulnerable: A story of telematic failures," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[10] S. Woo, H. J. Jo, and D. H. Lee, "A practical wireless attack on the connected car and security protocol for in-vehicle can," *IEEE Transactions on intelligent transportation systems*, vol. 16, no. 2, pp. 993–1006, 2014.

[11] C. Miller and C. Valasek, "A survey of remote automotive attack surfaces," *black hat USA*, vol. 2014, p. 94, 2014.

[12] S. Mazloom, M. Rezaeirad, A. Hunter, and D. McCoy, "A security analysis of an in-vehicle infotainment and app platform.," in *WOOT*, 2016.

[13] H. Wen, Q. A. Chen, and Z. Lin, "Plug-n-pwned: Comprehensive vulnerability analysis of obd-ii dongles as a new over-the-air attack surface in automotive iot," in *29th USENIXSecurity Symposium (USENIX Security 20)*, 2020, pp. 949–965.

[14] Y. Lee, S. Woo, J. Lee, Y. Song, H. Moon, and D. H. Lee, "Enhanced android app-repackaging attack on in-vehicle network," *Wireless Communications and Mobile Computing*, vol. 2019, 2019.

[15] C. Miller and C. Valasek, "Adventures in automotive networks and control units," *Def Con*, vol. 21, pp. 260–264, 2013.

[16] T. Telegraph, *Thieves are hacking into cars through their headlights, experts warn*, https://www.telegraph.co.uk/business/2023/04/09/thieves-hacking-cars-through-headlights/, 2023.

[17] S. Kulandaivel, S. Jain, J. Guajardo, and V. Sekar, "Cannon: Reliable and stealthy remote shutdown attacks via unaltered automotive microcontrollers," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2021, pp. 195–210.

[18] A. de Faveri Tron, S. Longari, M. Carminati, M. Polino, and S. Zanero, "Canflict: Exploiting peripheral conflicts for data-link layer attacks on automotive networks," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 711–723.

[19] R. Bhatia, V. Kumar, K. Serag, Z. B. Celik, M. Payer, and D. Xu, "Evading voltage-based intrusion detection on automotive can," *Network and Distributed System Security Symposium (NDSS)*, 2021.

[20] K.-T. Cho and K. G. Shin, "Error handling of in-vehicle networks makes them vulnerable," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 1044–1055.

[21] A. Palanca, E. Evenchick, F. Maggi, and S. Zanero, "A stealth, selective, link-layer denial-of-service attack against automotive networks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017, pp. 185–206.

[22] K. Iehira, H. Inoue, and K. Ishida, "Spoofing attack using bus-off attacks against a specific ECU of the CAN bus," in *IEEE Annual Consumer Communications Networking Conference (CCNC)*, 2018, pp. 1–4.

[23] P.-S. Murvay and B. Groza, "Dos attacks on controller area networks by fault injections from the software layer," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017, pp. 1–10.

[24] K. T. Cho and K. G. Shin, "Fingerprinting electronic control units for vehicle intrusion detection," in *USENIX Security Symposium*, 2016, pp. 911–927.

[25] S. Kulandaivel, T. Goyal, A. K. Agrawal, and V. Sekar, "Canvas: Fast and inexpensive automotive network mapping," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 389–405.

[26] K. T. Cho and K. G. Shin, "Viden: Attacker identification on in-vehicle networks," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 1109–1123.

[27] M. Kneib and C. Huth, "Scission: Signal characteristic-based sender identification and intrusion detection in automotive networks," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 787–800.

[28] M. Foruhandeh, Y. Man, R. Gerdes, *et al.*, "SIMPLE: Single-frame based physical layer identification for intrusion detection and prevention on in-vehicle networks," in *Annual Computer Security Applications Conference (ACSAC)*, 2019, pp. 229–244.

[29] M. Kneib, O. Schell, and C. Huth, "EASI: Edge-based sender identification on resource-constrained platforms for automotive networks," in *Network and Distributed System Security Symposium (NDSS)*, 2020, pp. 1–16.

[30] S. U. Sagong, X. Ying, A. Clark, *et al.*, "Cloaking the clock: Emulating clock skew in controller area networks," in *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2018, pp. 32–42.

[31] F. Yang, "A bus off case of CAN error passive transmitter," *EDN Technical paper*, 2009.

[32] T. Dagan and A. Wool, "Parrot, a software-only anti-spoofing defense system for the CAN bus," *ESCAR EUROPE*, 2016.

[33] D. Souma, A. Mori, H. Yamamoto, and Y. Hata, "Counter attacks for bus-off attacks," in *International Conference on Computer Safety, Reliability, and Security*, Springer, 2018, pp. 319–330.

[34] M. Takada, Y. Osada, and M. Morii, "Counter attack against the bus-off attack on CAN," in *Asia Joint Conference on Information Security (AsiaJCIS)*, 2019, pp. 96–102.

[35] B. Groza and P. Murvay, "Security solutions for the controller area network: Bringing authentication to in-vehicle networks," *IEEE Vehicular Technology Magazine*, vol. 13, no. 1, pp. 40–47, 2018.

[36] Q. Hu and F. Luo, "Review of secure communication approaches for in-vehicle network," *International Journal of Automotive Technology*, vol. 19, no. 5, pp. 879–894, 2018.

[37] I. Pekaric, C. Sauerwein, and M. Felderer, "Applying security testing techniques to automotive engineering," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, 2019, pp. 1–10.

[38] K. Serag, R. Bhatia, V. Kumar, Z. B. Celik, and D. Xu, "Exposing new vulnerabilities of error handling mechanism in can," *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[39] K. Serag, V. Kumar, Z. B. Celik, R. Bhatia, M. Payer, and D. Xu, "Attacks on can error handling mechanism," *International Workshop on Automotive and Autonomous Vehicle Security (AutoSec 2022)*, 2022.

[40] C. AUTOSAR, "Specification of secure onboard communication," *AUTOSAR CP Release*, vol. 4, no. 1, 2017.

[41] S. Nürnberger and C. Rossow, "–vatican–vetted, authenticated can bus," in *International Conference on Cryptographic Hardware and Embedded Systems*, Springer, 2016, pp. 106–124.

[42] B. Groza, S. Murvay, A. Van Herrewege, and I. Verbauwhede, "Libra-can: A lightweight broadcast authentication protocol for controller area networks," in *International Conference on Cryptology and Network Security*, Springer, 2012, pp. 185–200.

[43] A. Van Herrewege, D. Singelee, and I. Verbauwhede, "Canauth-a simple, backward compatible broadcast authentication protocol for can bus," in *ECRYPT Workshop on Lightweight Cryptography*, vol. 2011, 2011, p. 20.

[44] O. Hartkopp and R. M. SCHILLING, "Message authenticated can," in *Escar Conference, Berlin, Germany*, 2012.

[45] A. Hazem and H. Fahmy, "Lcap-a lightweight can authentication protocol for securing in-vehicle networks," in *10th escar Embedded Security in Cars Conference, Berlin, Germany*, vol. 6, 2012, p. 172.

[46] G. Bella, P. Biondi, G. Costantino, and I. Matteucci, "Toucan: A protocol to secure controller area network," in *Proceedings of the ACM Workshop on Automotive Cybersecurity*, 2019, pp. 3–8.

[47] M. D. Pesé, J. W. Schauer, J. Li, and K. G. Shin, "S2-can: Sufficiently secure controller area network," 2021.

[48] A.-I. Radu and F. D. Garcia, "Leia: A lightweight authentication protocol for can," in *European Symposium on Research in Computer Security*, Springer, 2016, pp. 283–300.

[49] K. Han, A. Weimerskirch, and K. G. Shin, "A practical solution to achieve real-time performance in the automotive network by randomizing frame identifier," *Proc. Eur. Embedded Secur. Cars (ESCAR)*, pp. 13–29, 2015.

[50] S. Woo, D. Moon, T.-Y. Youn, Y. Lee, and Y. Kim, "Can id shuffling technique (cist): Moving target defense strategy for protecting in-vehicle can," *IEEE Access*, vol. 7, pp. 15 521–15 536, 2019.

[51] Q. Wang and S. Sawhney, "Vecure: A practical security framework to protect the can bus of vehicles," in *2014 International Conference on the Internet of Things (IOT)*, IEEE, 2014, pp. 13–18.

[52] M. Müter, A. Groll, and F. C. Freiling, "A structured approach to anomaly detection for in-vehicle networks," in *Sixth International Conference on Information Assurance and Security (IAS)*, 2010, pp. 92–98.

[53] S. Kim *et al.*, "Shadowauth: Backward-compatible automatic CAN authentication for legacy ECUs," in *the ACM on Asia Conference on Computer and Communications Security*, 2022.

[54] T. Hoppe, S. Kiltz, and J. Dittmann, "Security threats to automotive CAN networks–practical examples and selected short-term countermeasures," *Reliability Engineering & System Safety*, vol. 96, no. 1, pp. 11–25, 2011.

[55] H. M. Song, H. R. Kim, and H. K. Kim, "Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network," in *International Conference on Information Networking (ICOIN)*, 2016, pp. 63–68.

[56] A. Taylor, N. Japkowicz, and S. Leblanc, "Frequency-based anomaly detection for the automotive CAN bus," in *World Congress on Industrial Control Systems Security (WCICSS)*, 2015, pp. 45–49.

[57] C. Young, H. Olufowobi, G. Bloom, and J. Zambreno, "Automotive intrusion detection based on constant CAN message frequencies across vehicle driving modes," in *Proceedings of the ACM Workshop on Automotive Cybersecurity*, 2019, pp. 9–14.

[58] W. Choi, K. Joo, H. J. Jo, *et al.*, "VoltageIDS: Low-level communication characteristics for automotive intrusion detection system," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 2114–2129, 2018.

[59] B. Groza, L. Popa, and P.-S. Murvay, "Incanta-intrusion detection in controller area networks with time-covert authentication," in *Security and Safety Interplay of Intelligent Software Systems*, Springer, 2018, pp. 94–110.

[60] B. Groza, L. Popa, and P.-S. Murvay, "Canto-covert authentication with timing channels over optimized traffic flows for can," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 601–616, 2020.

[61] X. Ying, G. Bernieri, M. Conti, and R. Poovendran, "Tacan: Transmitter authentication through covert channels in controller area networks," in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, 2019, pp. 23–34.

[62] S. Vanderhallen, J. Van Bulck, F. Piessens, and J. T. Mühlberg, "Robust authentication for automotive control networks through covert channels," *Computer Networks*, vol. 193, p. 108 079, 2021.

[63] A. Humayed, F. Li, J. Lin, and B. Luo, "Cansentry: Securing can-based cyber-physical systems against denial and spoofing attacks," in *European Symposium on Research in Computer Security*, Springer, 2020, pp. 153–173.

[64] B. Groza, L. Popa, P.-S. Murvay, Y. Elovici, and A. Shabtai, "Canary - a reactive defense mechanism for controller area networks based on active relays," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[65] A. Humayed and B. Luo, "Using id-hopping to defend against targeted dos on can," in *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles*, 2017, pp. 19–26.

[66] S. Ding, T. Zhao, R. Kurachi, and G. Zeng, "Id hopping can controller design with obfuscated priority assignment," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, IEEE, 2018, pp. 94–99.

[67] I. O. for Standardization (ISO), "Road Vehicles Controller area network (CAN)," *Part 1: Data link layer and physical signalling*, vol. ISO-11898-1:2015,

[68] K. Tindell and A. Burns, "Guaranteeing message latencies on control area network (can)," in *Proceedings of the 1st International CAN Conference*, Citeseer, 1994.

[69] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (can) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[70] R. I. Davis, S. Kollmann, V. Pollex, and F. Slomka, "Schedulability analysis for controller area network (can) with fifo queues priority queues and gateways," *Real-Time Systems*, vol. 49, no. 1, pp. 73–116, 2013.

[71] M. Di Natale, H. Zeng, P. Giusto, and A. Ghosal, *Understanding and using the controller area network communication protocol: theory and practice*. Springer Science & Business Media, 2012.

[72] K. Tindell, A. Burns, and A. Wellings, "Calculating controller area network (can) message response times," *IFAC Proceedings Volumes*, vol. 27, no. 15, pp. 35–40, 1994.

[73] K. Tindell, A. Burns, and A. J. Wellings, "Calculating controller area network (can) message response times," *Control engineering practice*, vol. 3, no. 8, pp. 1163–1169, 1995.

[74] W. YONG, "A scheduling algorithm for can bus," 2004.

[75] I. O. for Standardization (ISO), "Road Vehicles Controller area network (CAN)," *Part 2: High-speed medium access unit*, vol. ISO-11898-2:2016,

[76] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, "Chaskey: An efficient mac algorithm for 32-bit microcontrollers," in *International Conference on Selected Areas in Cryptography*, Springer, 2014, pp. 306–323.

[77] R. Kurachi, Y. Matsubara, H. Takada, N. Adachi, Y. Miyashita, and S. Horihata, "Cacan-centralized authentication system in can (controller area network)," in *14th Int. Conf. on Embedded Security in Cars (ESCAR 2014)*, 2014.

[78] J. D. Eckhardt, T. E. Donofrio, and K. Serag, *Multiple security level monitor for monitoring a plurality of mil-std-1553 buses with multiple independent levels of security*, US Patent 10,685,125, Jun. 2020.

[79] J. D. Eckhardt, T. E. Donofrio, and K. Serag, *System and method of monitoring data traffic on a mil-std-1553 data bus*, US Patent 10,467,174, Nov. 2019.

[80] J. D. Eckhardt, T. E. Donofrio, and K. Serag, *Bus data monitor*, US Patent 10,691,573, Jun. 2020.

[81] K. Serag *et al.*, "Zbcan: A zero-byte can defense system," *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[82] I. O. for Standardization (ISO), "Road vehicles FlexRay communications system," *Parts:1-5*, vol. ISO 17458:2013,

[83] R. Bosch, "Can with flexible data-rate specification," *Robert Bosch GmbH, Stuttgart*, 2012.

[84] I. O. for Standardization (ISO), "Road vehicles controller area network (can)," *ISO 11898-4:2004*, vol. Part 4: Time-triggered communication,

[85] M. Alliance, "MIPI A-PHY Long-reach SerDes," *v1.1*, vol. December 2021,

[86] A. S. Alliance, "ASA Overview and a Proof-of-Concept ASA Transceiver," *Rev 1.01*, vol. 2020,

[87] I. S. for Ethernet, "Amendment 1: Physical Layer Specifications and Management Parameters for 100 Mb/s Operation over a Single Balanced Twisted Pair Cable (100BASE-T1)," *IEEE Computer Society*, vol. IEEE Std 802.3bw:2015,

[88] I. S. for Ethernet, "Amendment 4: Physical Layer Specifications and Management Parameters for 1 Gb/s Operation over a Single Twisted-Pair Copper Cable," *IEEE Computer Society*, vol. IEEE Std 802.3bp:2016,

[89] Mobility Forsights, *Global electric vehicle wiring harness market*, https://mobilityforesights.com/product/electric-vehicle-wiring-harness-market/.

[90] Markets and Markets, *Automotive wiring harness market*, https://www.marketsandmarkets.com/Market-Reports/automotive-wiring-harness-market-170344950.html.

[91] I. S. for Ethernet, "Amendment 5: Physical Layer Specifications and Management Parameters for 10 Mb/s Operation and Associated Power Delivery over a Single Balanced Pair of Conductors," *IEEE Computer Society*, vol. IEEE Std 802.3cg:2019,

[92] C. in Automation, "Controller Area Network Extra Long (CAN XL)," *CiA 610*, vol. 1-3,

[93] A. Abdo, S. M. B. Malek, Z. Qian, Q. Zhu, M. Barth, and N. Abu-Ghazaleh, "Application level attacks on connected vehicle protocols," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 459–471.

[94] S. Hu, Q. A. Chen, J. Sun, Y. Feng, Z. M. Mao, and H. X. Liu, "Automated discovery of denial-of-service vulnerabilities in connected vehicle protocols," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3219–3236.

[95] B. Brecht *et al.*, "A security credential management system for v2x communications," *IEEE Transactions on Intelligent Transportation Systems*, vol. 19, no. 12, pp. 3850–3871, 2018.

[96] Y. Zhu, C. Miao, T. Zheng, F. Hajiaghajani, L. Su, and C. Qiao, "Can we use arbitrary objects to attack lidar perception in autonomous driving?" In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1945–1960.

[97] Y. Cao *et al.*, "Adversarial sensor attack on lidar-based perception in autonomous driving," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 2267–2281.

[98] R. S. Hallyburton, Y. Liu, Y. Cao, Z. M. Mao, and M. Pajic, "Security analysis of {camera-lidar} fusion against {black-box} attacks on autonomous vehicles," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1903–1920.

[99] B. Nassi, Y. Mirsky, D. Nassi, R. Ben-Netanel, O. Drokin, and Y. Elovici, "Phantom of the adas: Securing advanced driver-assistance systems from split-second phantom attacks," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 293–308.

[100] Y. Cao, S. H. Bhupathiraju, P. Naghavi, T. Sugawara, Z. M. Mao, and S. Rampazzi, "You cant see me: Physical removal attacks on lidar-based autonomous vehicles driving frameworks," *arXiv eprint archive*, 2022.

[101] Y. Zhu, C. Miao, F. Hajiaghajani, M. Huai, L. Su, and C. Qiao, "Adversarial attacks against lidar semantic segmentation in autonomous driving," in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, 2021, pp. 329–342.

[102] R. Quinonez, J. Giraldo, L. Salazar, E. Bauman, A. Cardenas, and Z. Lin, "{Savior}: Securing autonomous vehicles with robust physical invariants," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 895–912.

[103] K. Fu and W. Xu, "Risks of trusting the physics of sensors," *Communications of the ACM*, vol. 61, no. 2, pp. 20–23, 2018.

[104] S. Engineering, *Cybersecurity risks of automotive ota*, https://semiengineering.com/cybersecurity-risks-of-automotive-ota/.

A. EVALUATION OF STS

A.1 Learning Victim's Recovery Behavior

On the testbed, two ECUs were set up to implement a fixed penalty interval recovery model with a $35ms$ interval. Two other nodes were set up to implement the bare minimum model. Using SFBO, we were able to suppress all nodes, one at a time, as shown in Fig. 3.3.

At recovery, all nodes transmitted the attacked message as their first recovery message. However, only nodes that implemented the *fixed interval* time recovery model sent trailing messages. ECUs implementing the *bare minimum model* only sent the attacked message. This could be explained by the fact that at the testbed's busload of $\approx 15\%$, the average recovery interval of the bare minimum nodes was $\approx 4ms$, a period too short for another message to be buffered since the period for the fastest-transmitting ID for all ECUs was $10ms$.

For both of the ECUs implementing the *fixed interval* model, we tried attacking the IDs with the shortest period, and the IDs with the second shortest period. In both cases, and in both ECUs, the trailing messages had the ID with the shortest period in the ECU. However, in both cases, the first recovery message was the same message that was attacked.

On the vehicle, we evaluated SFBO on the four mapped ECUs. To ensure the ECUs truly transitioned to the *bus off* state, we recorded the traffic after every attack and observed the lack of any IDs that belong to the mapped ECU. This also validated our mapping results. On all ECUs, the first recovery message was the attacked message. Additionally, all ECU recoveries included trailing messages. The time recovery model for EBCM and BCM was identified as *sequenced intervals*. For the TCM and ECM, it was identified as *random*.

One challenge was identifying the *Optimum ID*. Looking at Table A.1, we notice that EBCM (ECU-1), has three IDs with the shortest period being $9ms$, TCM (ECU-3) has two IDs with a $12.5ms$ period, and ECM (ECU-4) has two IDs with a $12.5ms$ period. To pick the *optimum ID* for EBCM, we attacked it at IDs: $0x0C1, 0x0C5$ and $0x1E5$. In all cases, the trailing messages were of ID $0x0C1$. Hence, $0x0C1$ was selected as the optimum message for BCM. Similarly, $0x0F9$ and $0x0C9$ were selected for TCM and ECM, respectively.

Table A.1. Network map of ExpVehicle

ECU	ID	Period (ms)	ECU	ID	Period (ms)	ECU	ID	Period (ms)
ECU 1	C5	9	ECU 2	F1	10	ECU 3	199	12.5
	C1	9		1E1	30		F9	12.5
	1E5	9		1F3	33		19D	25
	1C7	18		1F1	100		1F5	25
	1CD	18		134	100		4C9	500
	1E9	18		12A	100		77F	1000
	184	18		3C9	100	ECU 4	C9	12.5
	334	18		3F1	233		191	12.5
	2F9	48		4E1	1000		1C3	25
	348	48		771	1000		1A1	25
	34A	48		4E9	1000		2C3	50
	17D	99		138	1000		3C1	100
	17F	99		514	1000		3E9	100
	773	1000		52A	1000		3D1	100
	500	1000		120	5000		3FB	250
							3F9	250
							4D1	500
							4C1	500
							4F1	1000
							772	1000

Overall Average Transmission Interval for ECU

ECU	Period (ms)
ECU 1	1.5
ECU 2	4.6
ECU 3	4.1
ECU 4	3.3

.

A.2 Recovery Prevention

On the testbed, ECU-1 and ECU-4 had a *bare minimum* recovery model. Hence, their recovery estimation and prevention was done by observing the number of 11 recessive-bit-instances and relaunching SFBO around the 128^{th} instance. On the other hand, ECU-2 and ECU-3 had a *fixed interval* model, with an identified recovery interval of $50ms$. Therefore, their recovery were estimated by starting a timer, and relaunching SFBO exactly when $50ms$ elapsed as described in Sec. 3.4.4. We were able to achieve an S_{rate} of 100% for at least $10s$ on all ECUs. After running the attack for 30 minutes, the average S_{rate} remained above 99.99%.

On the vehicle, EBCM and BCM have a sequenced recovery model. We used the ramp up attack shown in Fig. 3.9, to identify their sequences and prevent their recovery. As shown in Table 3.2, we identified 21 sequences for EBCM, and 13 for BCM, achieving maximum suppression periods of 2.38s and 1.42s, and average S_{rate} of 97.5% and 91.4%, respectively.

For the ECM and the TCM, we treated their recovery model as random since we could not identify any clear recovery pattern. Hence, we attacked the trailing message as described in Fig. 3.10, with IDs $0x0C9$, and $0c0F9$ selected as the *optimum IDs* for the ECM and the TCM, respectively. By attacking the trailing message, we were able to achieve maximum suppression periods of 3.51s and 1.38s, and average S_{rate} of 85%, and 83% for the TCM and the ECM, respectively.

As mentioned earlier, when attacking an ECU's optimum ID, the first trailing message will usually have the same ID. However, ECUs that have multiple IDs with similar, short periods will sometimes send other IDs in rare instances. This is the case with IDs: $0x0C9$ and $0x191$, and $0x0F9$ and $0x199$, in the ECM and TCM, respectively. When this happens, recovery prevention that relies on attacking the trailing message will fail, and the attacker will have to synchronize, re-launch SFBO, and proceed to prevent victim recovery again. This explains the slightly lower S_{rate} for ECM and TCM when compared to EBCM and BCM.

B. ZBCAN EVALUATION DETAILS

B.1 Evaluation Datasets

Table B.1 shows the messages of ExpVehicle. It also shows their theoretical WCRTs under four different settings. The first is without ZBCAN applied. The rest are after running Alg. 4 to find the optimum grouping and to calculate the WCRT for each group under three different $\|IBNSpan\|$ $settings$. Table B.2 shows the messages we used for our scalability evaluation. In the table, (G) refers to the priority group assignment, (Type) specifies whether the message is periodic or aperiodic, and (R) refers to the theoretical WCRT. In both tables, we used a queueing jitter $J_m = 750\ \mu s$. This number is based on the empirical observation of our testbed.

B.2 Measuring $\mathbf{d}_{skew}$

d_{skew} is related to the clock skews of every ECU, the clock skews of their CAN controllers, the ISR of each *agent*, its timer settings, and age. To measure d_{skew} of the system, several methods could be used. In our system, we measured it empirically. We ran the *Span Scan False Positive Test*, in which we connect a traffic source, sending a *reference message* every $6\ ms$. Next, we connect the first test ECU and set it up to send a message after every *reference message* with an *ascending* IBN between $0\ b$ and $3000\ b$. This means that the test ECU sends the first message with an $IBN = 0$ b and the second with $IBN = 1b$ and so on until it reaches $3000\ b$, then it rolls over to $0b$ again. Meanwhile, the *officer* monitors the IBN of each message and verifies their order. Once a message has an unexpected IBN, it is flagged as a false positive. We repeated the scan for $5\ min$ for each ECU. For each ECU, we marked the smallest IBN value after which inaccuracies start to occur as $d_{skew(ECU)}$. For all ECUs, the smallest d_{skew} was $1189\ b$, and the largest d_{skew} was $1460\ b$. We select $d_{skew(sys)}$ as the smallest $d_{skew} = 1189\ b$ for all ECUs.

Table B.1. WCRTs and groups for ExpVehicle.

Without ZBCAN				With ZBCAN					
ID	Length (B)	Period (ţs)	WCRT (ţs)	(32,32,32) WCRT (ţs)		(64,64,64) WCRT (ţs)		(32,64,128) WCRT (ţs)	
C1	8	9000	1274	P0	5290	P0	6250	P0	5546
C5	8	9000	1536						
1E5	8	9000	6026						
F1	4	9976	1962						
199	8	12477	4348						
F9	8	12486	2224						
C9	7	12488	1778						
191	8	12495	4086						
1C7	7	17980	5356						
1CD	5	17981	5560						
1E9	8	17981	6288						
184	6	18025	3824						
1C3	8	24986	5114	P1	14538	P1	17090	P1	15398
19D	8	25005	4610						
1F5	8	25014	6958						
1A1	7	25016	4852						
334	2	29977	7530						
1E1	5	30113	5764						
1F3	2	33278	6696						
2F9	5	47939	7384						
348	4	47954	7714						
34A	4	47956	7898						
2C3	6	50025	7180						
17D	8	98891	3340						
17F	8	98920	3602						
1F1	8	99694	6550						
134	4	99841	2874						
12A	8	99842	2690			P2	49676	P2	43786
3C9	8	99848	8422						
3C1	8	99938	8160	P2	32002				
3E9	8	99988	8946						
3D1	8	100052	8684						
3F1	8	233192	9208						
3FB	2	249738	10586						
3F9	8	249805	10440						
4D1	8	499492	11372						
4C1	8	499713	10848						
4C9	8	499872	11110						
4E1	8	997963	11634						
773	7	998184	14562						
500	4	998189	12284						
771	7	998237	14078						
4E9	5	998391	11838						
138	5	998424	3078						
514	8	998942	12546						
52A	8	999229	13836						
4F1	8	999307	12100						
772	7	999347	14320						
77F	8	999751	14824						
120	5	4992122	2428						

Table B.2. Scalability evaluation dataset.

ID	Type	T (ms)	D (ms)	R (ms)	G	Safety Ratio	ID	Type	T (ms)	D (ms)	R (ms)	G	Safety Ratio
1	A	10	10	7.3	0	0.73	33	P	250	250	96	2	0.384
2	P	10	10	7.3	0	0.73	34	P	250	250	96	2	0.384
3	P	10	10	7.3	0	0.73	35	P	250	250	96	2	0.384
4	P	10	10	7.3	0	0.73	36	P	250	250	96	2	0.384
5	P	10	10	7.3	0	0.73	37	P	500	500	96	2	0.192
6	A	25	25	7.3	0	0.292	38	P	500	500	96	2	0.192
7	P	25	25	7.3	0	0.292	39	P	500	500	96	2	0.192
8	P	25	25	7.3	0	0.292	3A	P	500	500	96	2	0.192
9	P	25	25	7.3	0	0.292	3B	P	500	500	96	2	0.192
A	P	25	25	7.3	0	0.292	3C	P	500	500	96	2	0.192
B	P	25	25	7.3	0	0.292	3D	P	500	500	96	2	0.192
C	P	25	25	7.3	0	0.292	3E	P	500	500	96	2	0.192
D	P	25	25	7.3	0	0.292	3F	P	500	500	96	2	0.192
E	P	25	25	7.3	0	0.292	40	P	500	500	96	2	0.192
F	A	50	50	37.11	1	0.7422	41	P	500	500	96	2	0.192
10	P	50	50	37.11	1	0.7422	42	P	500	500	96	2	0.192
11	P	50	50	37.11	1	0.7422	43	P	500	500	96	2	0.192
12	P	50	50	37.11	1	0.7422	44	P	500	500	96	2	0.192
13	P	50	50	37.11	1	0.7422	45	P	500	500	96	2	0.192
14	P	50	50	37.11	1	0.7422	46	P	500	500	96	2	0.192
15	P	50	50	37.11	1	0.7422	47	P	500	500	96	2	0.192
16	P	50	50	37.11	1	0.7422	48	P	500	500	96	2	0.192
17	P	50	50	37.11	1	0.7422	49	P	500	500	96	2	0.192
18	P	100	100	37.11	1	0.3711	4A	P	500	500	96	2	0.192
19	P	100	100	37.11	1	0.3711	4B	P	500	500	96	2	0.192
1A	P	100	100	37.11	1	0.3711	4C	P	500	500	96	2	0.192
1B	P	100	100	37.11	1	0.3711	4D	P	500	500	96	2	0.192
1C	P	100	100	37.11	1	0.3711	4E	P	500	500	96	2	0.192
1D	P	100	100	37.11	1	0.3711	4F	P	500	500	96	2	0.192
1E	P	100	100	37.11	1	0.3711	50	P	500	500	96	2	0.192
1F	P	100	100	37.11	1	0.3711	51	P	500	500	96	2	0.192
20	P	100	100	37.11	1	0.3711	52	P	500	500	96	2	0.192
21	P	100	100	37.11	1	0.3711	53	P	500	500	96	2	0.192
22	P	100	100	37.11	1	0.3711	54	P	500	500	96	2	0.192
23	P	100	100	37.11	1	0.3711	55	P	500	500	96	2	0.192
24	P	100	100	37.11	1	0.3711	56	P	500	500	96	2	0.192
25	P	100	100	37.11	1	0.3711	57	P	500	500	96	2	0.192
26	P	100	100	37.11	1	0.3711	58	P	500	500	96	2	0.192
27	P	100	100	37.11	1	0.3711	59	P	1000	1000	96	2	0.096
28	P	100	100	37.11	1	0.3711	5A	P	1000	1000	96	2	0.096
29	P	100	100	37.11	1	0.3711	5B	P	1000	1000	96	2	0.096
2A	P	100	100	37.11	1	0.3711	5C	P	1000	1000	96	2	0.096
2B	P	100	100	37.11	1	0.3711	5D	P	1000	1000	96	2	0.096
2C	P	100	100	37.11	1	0.3711	5E	P	1000	1000	96	2	0.096
2D	P	100	100	37.11	1	0.3711	5F	P	1000	1000	96	2	0.096
2E	P	100	100	37.11	1	0.3711	60	P	1000	1000	96	2	0.096
2F	P	100	100	37.11	1	0.3711	61	P	1000	1000	96	2	0.096
30	P	100	100	37.11	1	0.3711	62	P	1000	1000	96	2	0.096
31	P	100	100	37.11	1	0.3711	63	P	1000	1000	96	2	0.096
32	P	100	100	37.11	1	0.3711	64	P	1000	1000	96	2	0.096

C. ZBCAN PERFORMANCE ANALYSIS

C.1 Agent's Overhead Analysis

Memory. To function properly, *agents* require a minimum amount of memory as follows: (1) $L_{preshared}\ bits$ for the preshared key. (2) $L_{session}\ b$ for the session key. (3) $L_{counter}$ * $N_{ids-agent}\ b$ for the counter value of every ID transmitted by the *agent*. (4) $L_{sequence}$ * $N_{ids-agent} * 2\ bits$ to hold the IBN sequence for each of the IDs transmitted by the *agent*. Note that we store two sequences for each ID (Sec. 4.3.3). (5) L_{index} * $N_{ids-agent}\ bits$ for the index of the next IBN within each sequence for each ID. Assuming $L_{preshared} = L_{session} = L_{counter} = L_{sequence}$, the minimum amount of memory required is: $(2 * L_{sequence}) + ((3 * L_{sequence}) + L_{index}) * N_{ids-agent}\ bits$.

Sequence Extension Processing Overhead. An *agent* performs a *sequence extension* operation every time a sequence for a specific message ID runs out. This means that a *sequence extension* operation for a specific ID happens every $L_{seq}/\log_2(\|PSpan\|)$ outgoing $messages$. This number should be multiplied by the average period of message transmission for all IDs in the ECU (P_{av}) to calculate the average time between extensions (P_{ext}). We use (d_{ext}) to refer to the time needed to perform the extension operation itself. We recommend picking a fast PRF to perform the extension and minimize the overhead ratio $O_{ext} = d_{ext}/P_{ext}$. In our evaluation (Sec. 4.6), we achieved $O_{ext} \approx 1.91\%$ on the *agent's* side.

C.2 Officer's Overhead Analysis

Memory. The minimum amount of memory required by the *officer* is as follows: (1) $L_{preshared} * N_{agents}\ b$ for the preshared keys of all agents. (2) $L_{session} * N_{agents}\ b$ for the session keys of all agents. (3) $L_{counter}$ * $N_{ids-system}\ b$ for the counter value of every ID in the system. (4) $L_{sequence}$ * $N_{ids-system} * 2\ b$ for every IBN sequence in the system. (5) L_{index} * $N_{ids-system}\ bits$ for the index of the next IBN within every sequence in the system. Assuming that we are using the same length for $L_{preshared} = L_{session} = L_{counter} = L_{sequence}$, the minimum amount of required memory is: $(2 * L_{sequence} * N_{agents}) + ((3 * L_{sequence}) + L_{index}) * N_{ids-system}\ bits$.

Sequence Extension Processing Overhead. The *officer* performs a *sequence extension* operation whenever a sequence for a specific message ID runs out, meaning, a *sequence extension* operation for a specific ID happens every $L_{seq}/\log_2(\|PSpan\|)$ observed $messages$. In our evaluation (Sec. 4.6), we achieved $O_{ext} \leq 0.018\%$ on the *officer's* side.