Exposing New Vulnerabilities of Error Handling Mechanism in CAN

Khaled Serag and Rohit Bhatia, *Purdue University;* Vireshwar Kumar, *Indian Institute of Technology Delhi;* Z. Berkay Celik and Dongyan Xu, *Purdue University*

https://www.usenix.org/conference/usenixsecurity21/presentation/serag

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Exposing New Vulnerabilities of Error Handling Mechanism in CAN

Khaled Serag*, Rohit Bhatia*, Vireshwar Kumar†, Z. Berkay Celik*, and Dongyan Xu*
*Purdue University, {kserag, bhatia13, zcelik, dxu}@purdue.edu
†Indian Institute of Technology Delhi, viresh@cse.iitd.ac.in

Abstract

Controller Area Network (CAN) has established itself as the main internal communication medium for vehicles. However, recent works have shown that error handling makes CAN nodes vulnerable to certain attacks. In the light of such a threat, we systematically analyze CAN's error handling and fault confinement mechanism to investigate it for further vulnerabilities. In this paper, we develop CANOX, a testing tool that monitors the behavior of a CAN node under different bus and error conditions, and flags conditions that cause an unexpected node behavior. Using CANOX, we found three major undiscovered vulnerabilities in the CAN standard that could be exploited to launch a variety of attacks. Combining the three vulnerabilities, we construct the Scan-Then-Strike Attack (STS), a multi-staged attack in which an attacker with no previous knowledge of the vehicle's internals maps the vehicle's CAN bus, identifies a safety-critical ECU, swiftly silences it, and persistently prevents it from recovering. We validate the practicality of STS by evaluating it on a CAN bus testbed and a real vehicle.

1 Introduction

Since its introduction by Bosch in 1986, the Controller Area Network (CAN) protocol has offered low cost and efficient solutions to major challenges in vehicular communications among electronic control units (ECUs), such as interference, priority management, decentralization, error handling, and fault confinement [2]. 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 [3, 17, 19, 20, 26]. 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 [15]. In this paper, 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 [4, 12, 18, 21].

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 a node *throughout its operation* [2]. 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 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 moves to 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* [4] 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.

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 paper, 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) *Pas-*

sive 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 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, perpetuating the node's stay in the *bus off* state. (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) [4], 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 [5,6,8,13,14,16]. Nevertheless, these works approached network mapping from a defense standpoint. Thus, they either required physical access and special equipment [6, 8, 13, 14], or used time-consuming learning techniques that worked only with periodic messages [5, 16], and proved to be evadable [1, 23]. 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 to construct a powerful and persistent attack, STS.
- Demonstrating the practical impact of the vulnerabilities by evaluating STS on a testbed as well as a real vehicle.

2 Background and Motivation

2.1 Normal CAN Operation

Architecture of a CAN Node. A CAN node 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 a voltage signal that is transmitted on the bus. We note that the application code cannot directly control the CAN controller for transmitting single bits on the bus, nor can it precisely control the transmission time of a message.

Bit Communication. 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. 1, 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.

Arbitration. CAN uses lossless bitwise arbitration to detect 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. 1. CAN controllers sense the bus as they transmit every bit. During arbitration, 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. After arbitration, if a controller sending a recessive bit senses that the bus is dominant, it stops the transmission and raises an error.

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

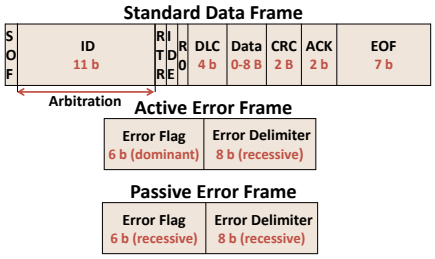


Figure 1: Different CAN frames.

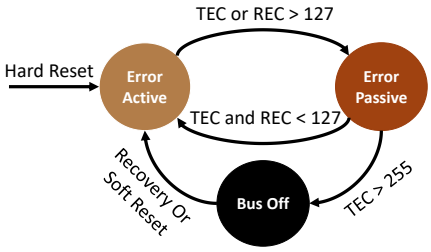


Figure 2: Error states in CAN.

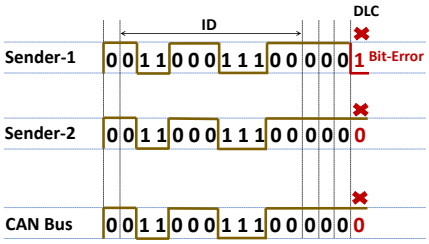


Figure 3: Producing a bit-error.

differ based on the error state of the node.

Error States. To provide fault confinement, CAN defines three error states as illustrated in Fig. 2.

(1) Error Active: A node by default is in this state. 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. 1). *Active error frames* override and terminate any ongoing transmission.

(2) Error Passive: 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. Further, on witnessing an error, the node transmits a *passive error frame*, consisting of 14 recessive bits (Fig. 1). Unlike the *active error frame*, a *passive error frame* is not observable on the bus and does not interrupt any ongoing transmission.

(3) *Bus Off:* 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.

Deliberate Packet Collisions. In [4], the authors propose a method to deliberately cause collisions on the bus using bit errors. The method includes two nodes simultaneously transmitting messages with the same ID, but with different contents, as shown in Fig. 3. To achieve simultaneous transmission with a specific message ID, the message arrival time for that ID should be known. Appendix A further explains how to synchronize two messages to cause a collision.

2.3 Threat Model

We assume that the attacker is an ECU node compromised by a remote attacker, capable of executing arbitrary software code on the node. Such abilities have been demonstrated in prior art [3, 9, 15, 17, 19, 20, 27]. The attacker does not have any prior knowledge of the in-vehicle network except for the *vehicle's make and model*. The attacker also needs to follow the specifications of the CAN controller hardware.

3 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

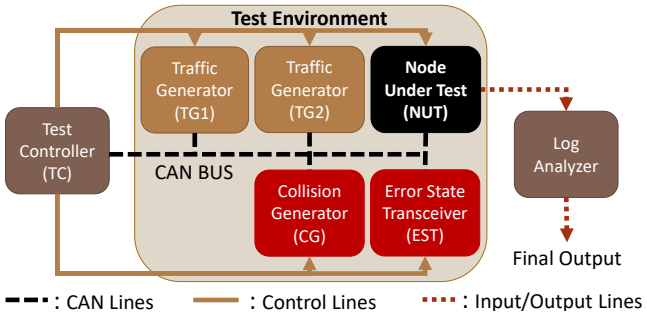


Figure 4: Architecture of CANOX.

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 [2], 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 define "unexpected" as a behavioral deviation from the *error active* behavior that exceeds a specific *threshold.* We describe the quantification of behavioral metrics in Sec. 3.1 and explain the thresholds required to find the deviations in Sec. 3.2. Using CANOX, we uncover three new fundamental vulnerabilities in CAN's error handling and fault confinement mechanism discussed in Sec. 4.

3.1 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. 4. 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 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.

Algorithm 1 `Test Controller Algorithm`

```
L ← {0%,...,100%}
P ← {lower, higher, mixed}
S ← {active, passive}
points = L × P × S
rounds ← {1,...,nrounds}
scenarios ← {single, collision, successive}
SDA ← Empty SD Array of Arrays
TA ← Empty TECC Array of Arrays
for s in scenarios do
    for p in points do
        for r in rounds do
            Turn off CG, TG, and NUT
            Set state of NUT
            Adjust TG to load and priority
            if (scen = collision) then
                Turn CG on
            Start NUT operation
            for ntrans in transmissions do
                Record SD and TECC
            Compute Average SD for the round
            Compute Average TECC for the round
        SDA_{s,p} ← Average SD across rounds
        TA_{s,p} ← Average TECC across rounds
    Pass SDA_s to Analyzer
    Pass TA_s to Analyzer
```

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 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 for further analysis if any *passive*

Algorithm 2 `Analyzer(SDArray, TECCArray, Th_SD, Th_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
```

metrics differ from the *active* metrics more than the specified thresholds. Threshold selection is detailed in Sec. 3.2.

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 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 [2] 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 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,

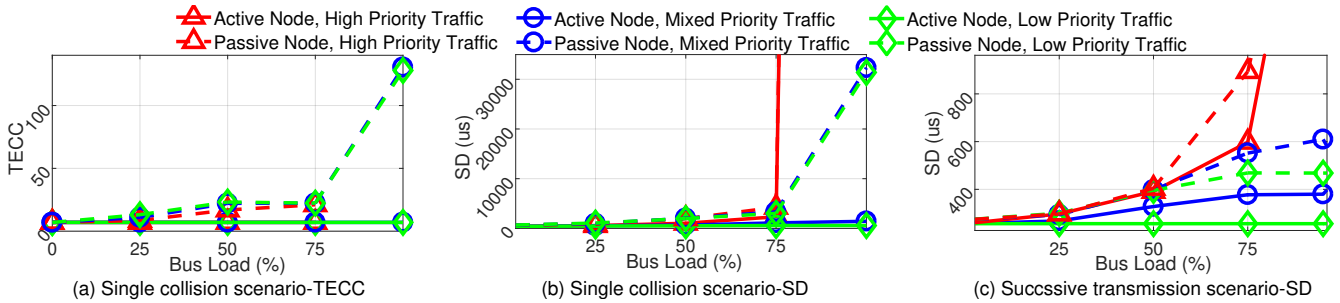


Figure 5: TEC change and standby delay values for the scenarios identified by CANOX as having an unexpected behavior.

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.

4 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. For the *successive transmission* scenario, CANOX 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. 5 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.

4.1 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. 5a, 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.*

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. These findings can be explained as follows.

A passive error frame consists of 14 *recessive bits* as shown in Fig. 1. However, the number of recessive bits at the end of a frame is 8, and the minimum bus idle time is 3 bit-periods [2]. 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* [2, 28]. 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.*

4.1.1 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 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. 6.

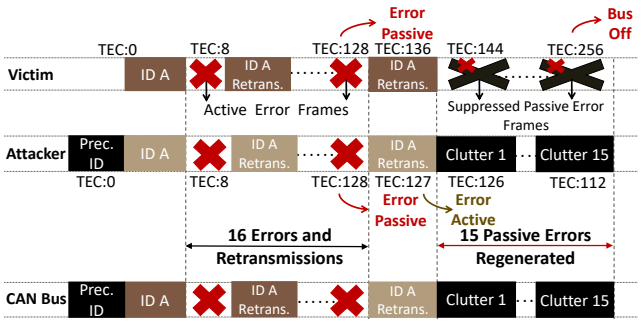


Figure 6: Illustration of the single frame bus off attack exploiting the passive error regeneration vulnerability.

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 paper, 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 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. 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) [4], 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. 4.2.1. In Sec. 6.3, we provide a comprehensive comparison between SFBO and OBA.

4.1.2 Exploit 2: Setting Victim's TEC

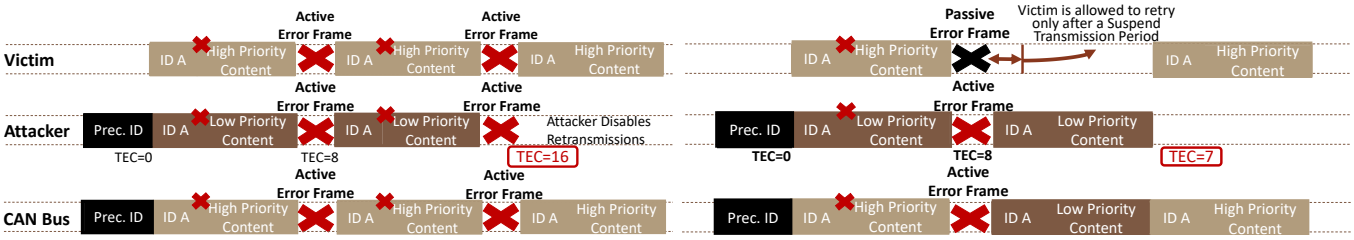
Exploit. The passive error regeneration vulnerability can be used to *easily* set the TEC of a victim node to a chosen value between 135 and 256. This can be done by following the first three steps of SFBO, but controlling the number of clutter messages (denoted by $\mathtt{N_{Clutter}}$) in Step 4, as discussed in Sec. 4.1.1. When $\mathtt{N_{Clutter}} = 15$, the victim falls into the *bus off* state. However, for any $\mathtt{N_{Clutter}} < 15$, the victim's TEC can be calculated as $\mathtt{TEC_{Victim}} = 135 + (8 * \mathtt{N_{Clutter}})$.

Impact. The ability to selectively set the victim's TEC value provides the attacker nearly full and immediate control of the victim's error states. The applications of such an exploit are versatile. For example, in Sec. 4.3.1, we explain how this exploit plays a critical role in identifying a message source.

4.2 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. 5b, we make three main observations. (1) The delay in the *passive* node is correlated with the bus load. (2) The SD curves in Fig. 5b are correlated with the TECC curves (Fig. 5a). (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*



(a) Behavior in the *error active* state.

(b) Behavior in the *error passive* state.

Figure 7: Identifying the error state of a victim by causing a collision and tracking TEC.

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.

4.2.1 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. 5.3.

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. 5.

4.3 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. 5c, 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.

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.

4.3.1 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. 4.1.2. 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 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, but 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. 7b, 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. 7a, 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. 5.1, 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.5.2. To the best of our knowledge, *this is the first network mapping technique to map aperiodic messages without using special hardware.*

5 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. 6.3. Lastly, its impact is persistent, as opposed to the previous volatile attacks.

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
```

5.1 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. 4.3.1. 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

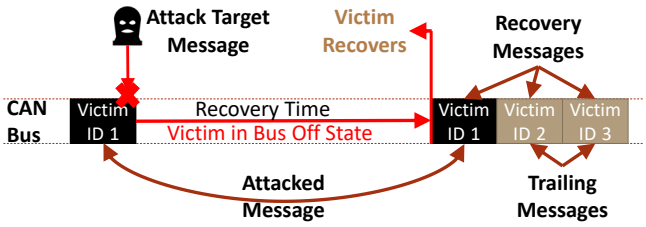


Figure 8: Illustration of the victim's recovery behavior.

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.

5.2 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), 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. 4.3.1, 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. 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.

5.3 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 paving the path for a persistent DoS attack. As such, STS needs to identify the victim's *recovery time* and the *recovery message*.

Recovery Messages. As discussed in Sec. 4.2.1, 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. 8. 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. 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.

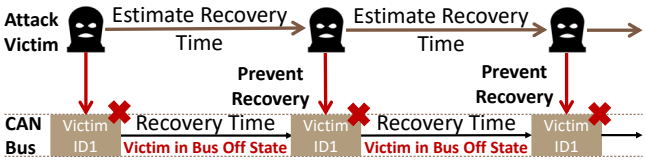


Figure 9: Demonstration of STS persistently preventing the victim's recovery from the *bus off* state.

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.

5.4 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 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. 4.2.1. (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. 9. Appendix B provides further details on Identifying and attacking different recovery models.

6 STS Evaluation

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) [4].

6.1 Evaluation Platforms

For in-depth analysis, the attack evaluation was carried out on a CAN bus testbed and a 2011 ExpCar[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. 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.

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

6.2 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, a small N decreased the calculation time but increased the error margin. To facilitate the use of preceded ID frame [4] 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.

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 100% accuracy. 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 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, $T_{ECU} = (2 * N_{ids} * (T_s + T_{avg})) + (t_{cool} * N_{ids}) + (T_{jitter})$. 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.

Table 1: Network mapping results for a 2011 ExpCar[1].

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)

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. 5.2. Table 1 shows the identification results. 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 35*ms* 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. 4.1.1. For all ECUs, the optimum attack ID was identified as 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, it was identified as the *random*. Additionally, for all ECUs, we were able to identify the optimum attack ID satisfying the two conditions mentioned in Sec. 5.3. Table 2 shows the optimum attack ID for each ECU.

Table 2: Suppression rates for different ECUs on ExpCar[1].

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. 5.4, we were able to achieve an S_{rate} of 100% for at least 10*s* 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 2, using the techniques described in Sec. 5.4, 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.

Table 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%

6.3 Comparing SFBO to OBA

We compare the impact of using OBA [4] 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. 10, in comparison to $\approx 5ms$ taken by SFBO, OBA required around 180*ms*, 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 210*ms* for ECUs 1, 2, 3 and 4, respectively.

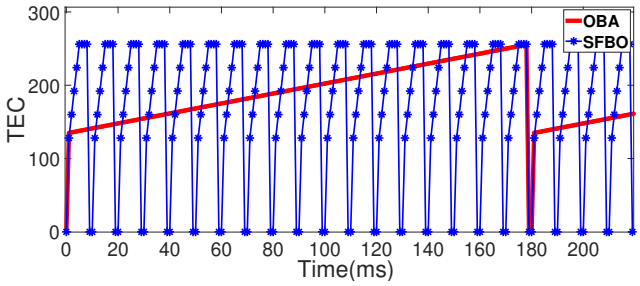


Figure 10: Swiftness of SFBO compared to the best case scenario of OBA.

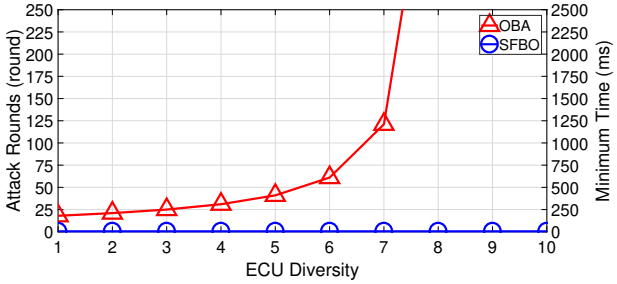


Figure 11: Illustrating the impossibility of OBA when ECU diversity exceeds 8.

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, the suppression rates changed for the ECUs that implement 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, 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 be 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 ECU 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. 11, 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.

7 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, where 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.

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. 4.3.1 could be used for reverse engineering purposes. Similarly, recent works [1] 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. 4.1.2 comes 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.

8 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.

9 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 [7, 24, 25] 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 [10, 11], 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.

10 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 [3, 17, 19, 20]. 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 [15]. Hence, it is imperative to develop frameworks that can methodically discover the full spectrum of vulnerabilities suffered by CAN under such scenarios [22]. 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 [4]. However, as shown in this paper, 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 [12, 18, 21]. 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 [5, 16]. However, their learning techniques were prone to inaccuracies and proved to be evadable [23]. Others suggested using voltage signatures of ECUs [6, 13] 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.

11 Conclusion

In this paper, 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.

Acknowledgments

We thank the anonymous reviewers and Dr. Flavio Garcia for their valuable comments and suggestions. This work was supported in part by the Office of Naval Research (ONR) under Grant N00014-18-1-2674. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] Rohit Bhatia, Vireshwar Kumar, Khaled Serag, Z Berkay Celik, Mathias Payer, and Dongyan Xu. Evading voltage-based intrusion detection on automotive CAN. *Network and Distributed System Security Symposium (NDSS)*, 2021.

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

[3] S. Checkoway, D. Mccoy, B. Kantor, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, pages 77–92, 2011.

[4] 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)*, pages 1044–1055, 2016.

[5] K.-T. Cho and K. G. Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *USENIX Security Symposium*, pages 911–927, 2016.

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

[7] Tsvika Dagan and Avishai Wool. Parrot, a software-only anti-spoofing defense system for the CAN bus. *ESCAR EUROPE*, 2016.

[8] 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.

[9] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. Fast and vulnerable: A story of telematic failures. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*, 2015.

[10] B. Groza and P. Murvay. Security solutions for the controller area network: Bringing authentication to in-vehicle networks. *IEEE Vehicular Technology Magazine*, 13(1):40–47, 2018.

[11] Q. Hu and F. Luo. Review of secure communication approaches for in-vehicle network. *International Journal of Automotive Technology*, 19(5):879–894, 2018.

[12] 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)*, pages 1–4, 2018.

[13] 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)*, pages 787–800, 2018.

[14] Marcel Kneib, Oleg Schell, and Christopher Huth. Easi: Edge-based sender identification on resource-constrained platforms for automotive networks. In *Proc. Netw. Distrib. Syst. Secur. Symp.*, pages 1–16, 2020.

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

[16] Sekar Kulandaivel, Tushar Goyal, Arnav Kumar Agrawal, and Vyas Sekar. Canvas: Fast and inexpensive automotive network mapping. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 389–405, 2019.

[17] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015:91, 2015.

[18] Pal-Stefan Murvay and Bogdan 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*, pages 1–10, 2017.

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

[20] 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.

[21] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano 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*, pages 185–206, 2017.

[22] Irdin Pekaric, Clemens Sauerwein, and Michael Felderer. Applying security testing techniques to automotive engineering. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, pages 1–10, 2019.

[23] 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)*, pages 32–42, 2018.

[24] Daisuke Souma, Akira Mori, Hideki Yamamoto, and Yoichi Hata. Counter attacks for bus-off attacks. In *International Conference on Computer Safety, Reliability, and Security*, pages 319–330. Springer, 2018.

[25] Masaru Takada, Yuki Osada, and Masakatu Morii. Counter attack against the bus-off attack on CAN. In *Asia Joint Conference on Information Security (AsiaJCIS)*, pages 96–102, 2019.

[26] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. Plug-n-pwned: Comprehensive vulnerability analysis of obd-ii dongles as a new over-the-air attack surface in automotive iot. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 949–965, 2020.

[27] Samuel Woo, Hyo Jin Jo, and Dong Hoon Lee. A practical wireless attack on the connected car and security protocol for in-vehicle can. *IEEE Transactions on intelligent transportation systems*, 16(2):993–1006, 2014.

[28] Fuyu Yang. A bus off case of CAN error passive transmitter. *EDN Technical paper*, 2009.

A Deliberate Packet Collisions

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 exactly at the expected arrival time. For a periodic message, this could be done by monitoring the message ID 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 [4], 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. 12, 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.

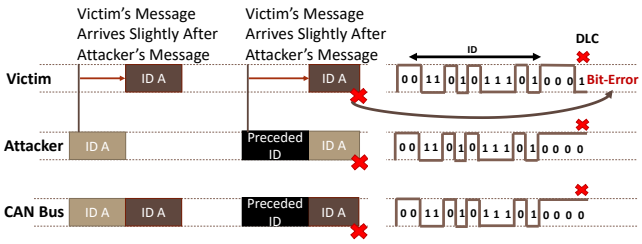


Figure 12: Enabling time synchronization between the attacker's and the victim's messages by using a preceded ID message to facilitate packet collisions on the CAN bus.

B Recovery Estimation and Prevention

B.1 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. 13.

Bare Minimum. If the time corresponds to 128 instances of 11 recessive bits, the model will be the bare minimum.

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 and letting it recover.

Sequenced Intervals. In a sequenced intervals model, the victim uses a different interval every time the recovery is

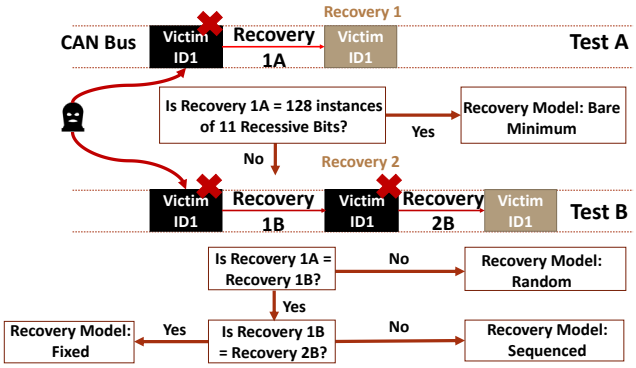


Figure 13: Determining victim's time recovery model

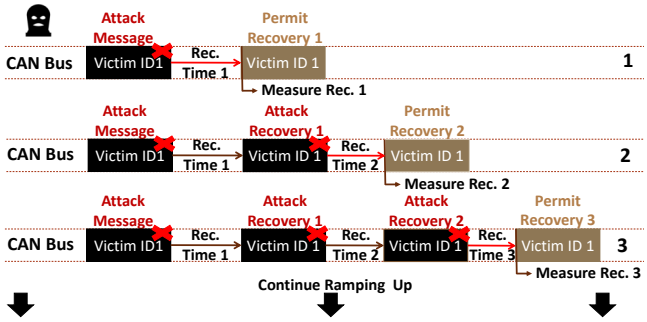


Figure 14: Ramping up suppression rate by learning more recovery sequences every iteration

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.

Random. If there is no observable pattern in the victim's recovery time, the attacker considers the model to be random.

B.2 Recovery Prevention

Since there exist multiple *time recovery models*, the recovery estimation method differs between models. We now explain how to estimate recovery time for each model.

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.

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

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. 14.

Random. There is no way to expect when the ECU with

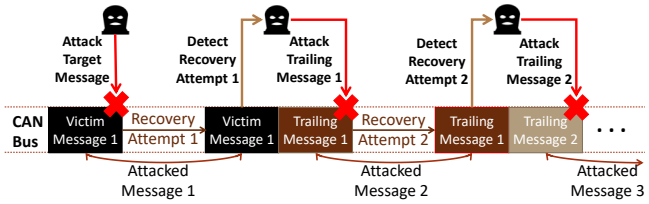


Figure 15: Suppressing victims with random recovery times by attacking trailing recovery messages.

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. 15. 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 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.

C Evaluation of STS

C.1 Victim Function Identification

On our 2011 ExpCar, 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 1.

C.2 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, by attacking a single message as shown in Fig. 6.

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 is because, at the testbed's bus-load 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 4, 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.

C.3 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. 5.4. 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. 14, to identify their sequences and prevent their recovery. As shown in Table 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, their recovery model was identified as random. Hence, we attacked the trailing message as described in Fig. 15, 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.

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

Table 4: Network map of a 2011 ExpCar[1].