Revisiting remote attack kill-chains on modern in-vehicle networks

Submitted in partial fulfillment of the requirements for
the degree of
Doctor in Philosophy
in
Electrical and Computer Engineering

Sekar Kulandaivel

B.S., Computer Science and Electrical Engineering, University of Maryland, Baltimore County

Carnegie Mellon University
Pittsburgh, PA

December 2021

Acknowledgements

I would like to express my sincere gratitude for all who have supported me along this journey over the years.

First, I want to thank my advisor, Vyas Sekar, for his unwavering support and belief in my success as he guided me through the Ph.D. process. During every meeting with him, I continue to be astounded at his brilliant insights and guidance, which have truly made me the researcher I am today. Vyas taught me how to convey the impact of my work and formulate complex problems into a clear and thoughtful presentation. His advice has opened many doors for me, and I will always be thankful for that. It has been a great privilege to have Vyas as my advisor.

I want to thank and acknowledge the support of my thesis committee members: Jorge Guajardo, Anuja Sonalker, Raj Rajkumar, and Lujo Bauer. I am grateful for the opportunity to work closely with Jorge and learn many lessons from him over the course of two internships. Jorge has given me a new appreciation for performing research that considers all of the involved stakeholders, and he has been an outstanding mentor who has helped me navigate the automotive industry. It is always a pleasure to work with Jorge. Prior to starting graduate school, I had the privilege of working with Anuja who introduced me to the excitement of automotive security. From hacking cruise control on a real vehicle to winning a Car Hacking Village competition at DEF CON, Anuja helped me craft many of the skills that have led to the success of my research. She always offered words of encouragement, and she has always been confident in my abilities. I truly appreciate the impact she has had on my career. From being a student in his class to eventually being the class TA, I have admired Raj for all of his lessons and contributions to the automotive field. He has always asked the most interesting questions to ensure that my research has the greatest impact it can, and I will always appreciate the opportunities I have had to work with him. I continue to be amazed at how he pictures my work and its potential long-term impact on the field. From my qualifying exam to my defense, I consider myself lucky to have had him involved at each stage of my Ph.D. process. While I have not had the pleasure of collaborating closely with Lujo, his constructive feedback and insights have helped me improve my thesis. I am grateful for his time and willingness to support me as a growing researcher. To all my committee members: it has been a real honor to have such incredible mentors along this journey.

I also want to acknowledge the many additional collaborators that I have worked with over the years: Tushar Goyal, Arnav Kumar Agrawal, Venkat Viswanathan, Shashank Sripad, and Shalabh Jain. I had a lot of fun with Tushar and Arnav as we tore apart the Prius for our class project and eventually my first paper. We taught each other as we worked together to navigate the challenges of vehicle networks, and I benefited immensely from their ideas and contributions. Working with Venkat and Shashank on electric vehicle security introduced me to the wider impact that my research could achieve, and I appreciate the many conversations we had on such interesting topics. Shalabh has been my most recent collaborator, and he has also been a great mentor. He has supported me in my pursuit of new opportunities to share my work, and he has openly shared his knowledge of the field to make me a stronger researcher. I am grateful for all of the collaboration that I have had and for the many lessons I have learned from them all.

The journey through my Ph.D. program came with an incredible amount of support from CyLab. I want to thank all of the CyLab staff and faculty for giving us students a wonderful experience. I am grateful for the student seminar opportunities and the many discussions with Cylab research groups. I received the best assistance any student could ask for from the administrative staff of CyLab and the ECE department, including Brigette Bernagozzi, Chelsea Mendenhall, Toni Fox, Jamie Scanlon, Brittany Frost, Nathan Snizaski, Karen Lindenfelser, and Ivan Liang.

I also want to thank my friends at CMU and outside of school for encouraging me and cheering me on. It has been a pleasure to meet my new friends, and I am grateful for my old friends. Two of the most important people in this journey are my parents. They have given me everything so that I may succeed in my dreams. They have and continue to support me unconditionally, and they have always believed in me. My sister has also played a major role in my success from when I was very young to even now. She has helped me grab every opportunity made available to me, and I hope to continue to follow in her footsteps. I also want to thank my in-laws for being my biggest advocates and for attending many of my major events. They never fail to cheer me on.

Finally, the most important person throughout this journey has been my wife, Katie Kulandaivel. She has walked with me through this entire journey. From when I got the letter of acceptance into the Ph.D. program to the day I submit this thesis, she has always been right by my side. She has kept me going from day one, and she has given me her all in supporting my dreams. I will forever be thankful for her and her support. She has shown me the most love I could hope for through every struggle and challenge, and I simply could not have done this without her.

The work presented in this thesis was funded in part by the PITAXVIII PITA award and the CNS-1564009 NSF IoT award. We gratefully acknowledge support from Technologies for Safe and Efficient Transportation (T-SET) University Transportation Center. This work was also supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Thesis Committee Members:

Vyas Sekar (Chair)

Jorge Guajardo

Anuja Sonalker

Raj Rajkumar

Lujo Bauer

Abstract

In-vehicle networks contain an increasing number of electronic control units (ECUs) with advanced electronics and wireless capabilities. Due to their critical role in vehicles, these ECUs are a prime target for remote adversaries as ECUs often communicate via the reliable but insecure Controller Area Network (CAN) protocol. By compromising just a single in-vehicle ECU, a remote adversary could manipulate safety-critical systems by simply injecting CAN messages. Prior work had demonstrated the feasibility and severity of real-world remote exploitation of in-vehicle ECUs, which brought this threat to the attention of automotive manufacturers/suppliers and global regulatory bodies. In response, the automotive industry has since developed defenses to secure the CAN bus against remote adversaries, and these defenses do well to detect and prevent known message injection attacks.

In this thesis, we argue that there remain unaddressed disconnects in the security design of modern in-vehicle networks. We develop an end-to-end attack "*kill-chain*" that demonstrates a series of exploited vulnerabilities in modern vehicles. Here, we envision an adversary that remotely compromises a non-safety-critical and wirelessly-connected ECU (e.g., infotainment) with the goal of controlling a safety-critical ECU (e.g., engine, transmission) while evading detection by modern network defenses. However, these defenses can still prevent our attacker from simply using the compromised ECU to inject critical CAN messages that disrupt the safety-critical ECU's functionality. Therefore, we aim to construct a kill-chain that can ultimately enable a remote adversary to gain control of a safety-critical ECU's software, opening the door to more advanced safety-critical attacks. By identifying disconnects that an adversary can exploit to build this kill-chain, we can inform defenses for next-generation vehicles with proposals of countermeasures that target these disconnects.

Our key contributions consist of building new attack classes, demonstrating attack feasibility on real vehicles, and proposing countermeasures for each stage of our kill-chain. First, to gauge an understanding of a victim network of ECUs and their transmission characteristics, our CANvas network mapper accurately identifies the source and destination ECU of a given CAN message and permits us to characterize ECU transmissions. Using this network knowledge, the CANnon disruption technique demonstrates how an adversary can target a victim ECU in the network and disrupt its CAN transmissions to the adversary's advantage. Finally, the CANdid authentication bypass leverages both CANvas and CANnon to successfully authenticate with a safety-critical victim ECU without access to its secret keys. To complete the kill-chain, we demonstrate how our three stages enable a remote adversary to download code to a victim ECU. Drawing from the vulnerabilities that enable our kill-chain, we propose practical countermeasures to detect and prevent our methods and discuss the lessons we learned to help identify potential vulnerabilities in a future automotive network design.

Contents

List of Figures

List of Tables

1 Introduction

As a prevailing mode of transportation in the United States, vehicles serve a major role in the country's critical infrastructure [1]. We depend on more than 14 million trucks to carry over 18 billions tons of freight, and we operate over 250 million passenger vehicles on our highways [2]. With the push to electrify vehicles [3], they will also play a role in energy infrastructure since vehicles account for over 20% of the total U.S. energy use [4]. In addition, the advent of connected and autonomous vehicles will further expand the impact of vehicles on U.S. critical infrastructure [5].

To ensure the safety of passengers and freight, vehicles utilize a complex network of electronic control units (ECUs) to control safety-critical systems, such as braking and engine control [6]. Unfortunately, prior work by Koscher et al. in 2010 demonstrated vulnerabilities that target such systems on this in-vehicle network [7]. The authors physically connected to a vehicle's network and injected malicious traffic that disabled the brakes and stopped the engine. However, as this work required physical access, these attacks were considered impractical and could not target this critical infrastructure at a large scale [8].

Where prior work highlighted the feasibility of a remote attack [8, 9], the work by Miller et al. in 2015 brought such an attack to reality [10]. The authors remotely exploited the infotainment ECU on a 2014 Jeep Cherokee and gained a foothold within the vehicle's network. Using this compromised ECU, the authors launched network attacks by simply injecting malicious traffic and disabled the brakes and stopped the engine (similar to the work by Koscher et al. [7]). This work was key in demonstrating both a remote attack and one that could be performed at a large scale, ultimately leading to a recall of 1.4 million vehicles [10].

In retrospect, it is understandable how in-vehicle networks eventually became vulnerable to a remote attack. In the 1970s, vehicles started to include electronics [11], and then, in 1986, vehicles had enough ECUs that Robert Bosch GmbH introduced a new in-vehicle network, the

Controller Area Network (CAN) [12], which is still widely used today. Until recently, vehicle design focused on reliability and safety for passengers and freight, thus security was not a concern. Then, we saw the demonstration of physical network attacks [7] followed by a real remote attack [10], which drew the attention of global regulatory bodies [13] and the automotive industry [14]. We have since seen a slew of state-of-the-art network defenses [15, 16, 17, 18, 19, 20, 21] that appear to do well against the adversary from the 2015 remote attack [10].

1.1 Pushing the boundaries of a remote attack

The remote attack by Miller et al. in 2015 [10] was just the first of several demonstrations of a remote compromise situation on real vehicles [22, 23, 24]. All of these works often focus on transmitting malicious traffic onto the in-vehicle network with the goal of impacting safety-critical systems. However, in this thesis, we determine how far a remote adversary can take their attack. In other words, we investigate a realistic worst-case scenario that should be considered when analyzing the risk of a remotely-compromised in-vehicle ECU. By exploring the true capabilities of a remote adversary, we can use our findings to pinpoint security gaps in modern vehicle implementations and then inform future defenses.

Where prior work on remote attacks focused on getting access to the in-vehicle network [10, 22, 23, 24], we aim to push the boundaries of such access by borrowing a popular concept from traditional network security: the attack kill-chain, an end-to-end attack consisting of multiple stages that an adversary must consecutively complete [25]. With access to the in-vehicle network in their attack, Miller et al. state that injecting malicious traffic to disable the brakes and kill the engine only succeed when the vehicle travels under 5-10 mph [10]. However, to bypass this limitation, a remote attacker already in the vehicle's network could remove or avoid this speed threshold by reprogramming the affected safety-critical ECU (e.g., reprogram the engine ECU to accept "kill engine" commands at any vehicle speed). Thus, we envision a remote attack that can achieve this level of control as a worst-case scenario.

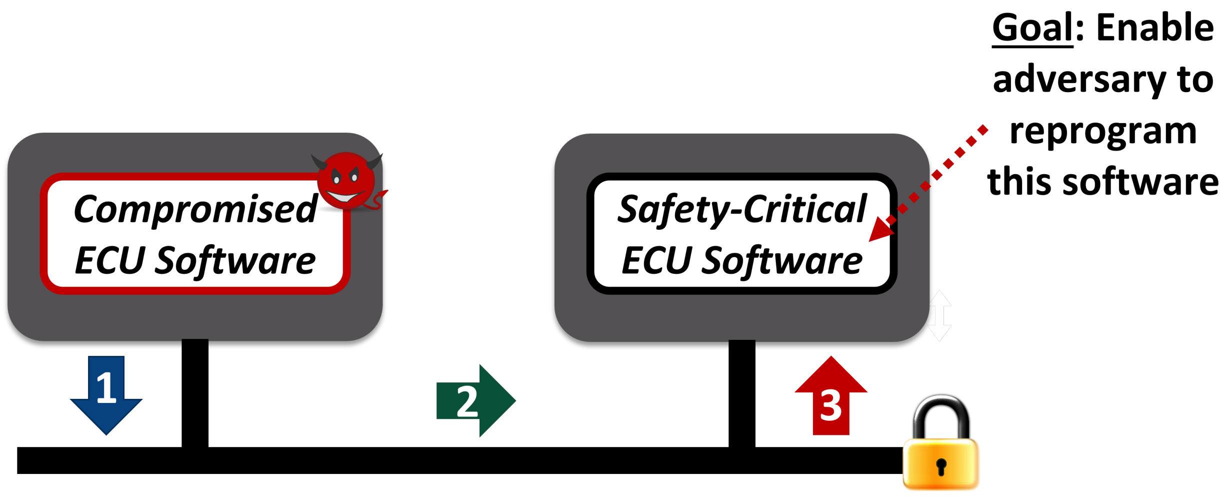

Goal: Enable adversary to reprogram this software
Compromised ECU Software
Safety-Critical ECU Software
1
2
3

Figure 1: To enable an adversary to reprogram a safety-critical ECU, we envision three adversary goals: (1) identify in-vehicle ECUs and their transmissions, (2) control transmissions of other ECUs, and (3) bypass authentication on a safety-critical ECU.

To investigate the potential for such a remote kill-chain as depicted in Figure 1, we set the starting point of this kill-chain to be a remote adversary that controls the software for a single in-vehicle ECU; as demonstrated in prior work, this is likely an infotainment or telematics ECU [10, 22, 23, 24]. Achieving the goal of enabling the remote adversary to reprogram another (likely safety-critical) ECU requires the adversary to complete several intermediary stages while deceiving state-of-the-art defenses. There are three stages that we focus on: (1) the adversary must assess a victim network by identifying participant ECUs and their impact on the network, (2) the adversary must demonstrate precise control over another ECU's transmissions, and (3) the adversary must bypass the authentication on a target safety-critical ECU to ultimately enable reprogramming capabilities on this ECU. By achieving this stronger capability at the hands of a remote adversary, we can demonstrate that such an adversary (i.e., the potential result of a single remotely-compromised ECU) is more of a concern than before even with state-of-the-art defenses in place.

1.2 Potential impact of a stronger remote adversary

As market research estimated that about 150 to 250 million connected cars (i.e., cars remotely linked to networks) were on the road in 2020 [26, 27, 28], a remote adversary will likely target the infotainment ECU or other ECUs in the vehicle with one or more wireless interfaces [9]. Additionally, as we pass more control functions from the driver to the vehicle for autonomous driving, the ECUs in a vehicle will be responsible for more safety-critical functionality [5]. As a result, with the addition of new connected vehicle technologies and more computer-controlled safety systems, modern vehicles will become increasingly vulnerable to remote exploits that aim to target these safety-critical systems [29]. Considering the number of recent remote attack demonstrations [10, 22, 23, 24], it is likely to expect future zero-day or day-one vulnerabilities on modern vehicles.

In addition to vehicles starting to include more remote interfaces, another aspect that aids the adversary is that vehicles are often replicas of each other when looking at network components and characteristics. Within a single make and model, the software and hardware is very similar (if not exactly the same) between different instances of vehicles. As a result, a vulnerability that relates to the *design* of the vehicle and its components can expose potentially thousands of vehicles to the same vulnerability. This thesis aims to focus on attacks that exploit design similarities that could maximize an attack's impact. Considering this threat of a remote adversary who could impact thousands of vehicles, it is critical that the industry does not underestimate such an adversary's capabilities. Thus, we should draw a model of a worst-case scenario that a remote adversary could achieve.

The scenario that we envision here is an adversary that can take control over a safety-critical ECU. Rather than relying on CAN message injection as done by prior work [10, 22], an adversary that could *reprogram* a vehicle's engine ECU (or any other safety-critical ECU) could significantly strengthen an adversary's capabilities. With control of a safety-critical ECU's firmware, the adversary could now access all of the ECU's inputs (from sensors, e.g., throttle position) and outputs (to actuators, e.g., fuel injectors) and could bypass any firmware constraints. For example, instead of only succeeding to disable the brakes at a low speed as seen in prior work [10], this adversary could reprogram the brake ECU and then disable the brakes at any speed by simply removing or bypassing safety-related program code. Additionally, if an adversary could perform this reprogramming, they could launch dormant attacks where the attack code sits on the safety-critical ECU until it is triggered to attack at a specific time. The authors of the Jeep hack even alluded to the "very interesting and scary" idea of a worm [10], which closely follows this concept.

1.3 Threat scope

To clarify our threat model, we envision a remote adversary who has already exploited a wirelessly-connected ECU and compromised its software with some exploit. Given the proprietary nature of ECU software, the adversary's techniques should not require any physical access to the victim vehicle or access to original ECU code from the manufacturer. We assume the adversarial capabilities are limited to *only* software manipulation and do not allow for direct *physical modifications or probing* to any of the vehicle's components. In this thesis, we perform all of our attacks on the victim vehicle using available software instructions. We follow the same assumptions of prior work, which assume that the adversary can modify the application software on an ECU and utilize any programming interfaces available to this software [30, 17, 15, 16].

In addition to the constraints placed on the adversary due to its remote access, our adversary also faces the challenge of attacking in the presence of modern defenses. For the attacks we present, we identify modern defenses as those that are implemented in modern vehicles and by manufacturers. To clarify, defenses that are less costly to implement (e.g., defenses that exist without additional hardware) are likely to be considered over more costly methods (e.g., adding new hardware). By identifying and leveraging properties that enable the kill-chain in ways that remain stealthy to network defenses, we can then identify practical countermeasures to better secure future in-vehicle networks. However, identifying these methods is challenging as the design of modern vehicles is largely proprietary and little information is available about a specific vehicle's design. Thus, we will need to leverage commonalities among network designs and identify vulnerabilities that either exist or remain due to choices in vehicle network design. As we leverage these vulnerabilities, we must ensure that our methods are robust when applied to real networks as any error during a real attack could increase the likelihood of the adversary getting caught.

1.4 Thesis contributions

As discussed above, a remote adversary in control of a single in-vehicle ECU that then reprograms another safety-critical ECU opens the door to high-impact attacks beyond attacks using malicious message injection. To determine if such an adversary could exploit their control of a single ECU to enable the reprogramming of a target ECU, we must investigate the necessary steps that could permit our adversary to achieve their goal. Due to the multiple layers involved between the software of the initial compromised ECU and the software of the second safety-critical ECU, our adversary will need to traverse many components of the vehicle network design. In addition, the adversary will need to evade modern defenses implemented on the network, which can exist at each layer of the network design. With this in mind, we motivate the need for an attack kill-chain approach, or a series of exploited vulnerabilities, that can identify existing gaps in modern automotive networks. This kill-chain can demonstrate how an attacker with remote control of a single (likely non-safety-critical) ECU can ultimately control a safety-critical ECU with little chance of detection and prevention by modern network defenses.

To enable this kill-chain, we design three stages of the attack for a remote adversary to perform as depicted in Figure 2: (1) reconnaissance to identify and characterize ECUs on the network, (2) disruption of typical network operation to control ECU transmissions, and (3) authentication bypass to enable reprogramming of safety-critical ECU. By successfully completing all three stages, we could now enable the adversary to download their attack code [1] to the critical

[1]Note that, in this work, we do not investigate the final attack payload as that is dependent on make and model

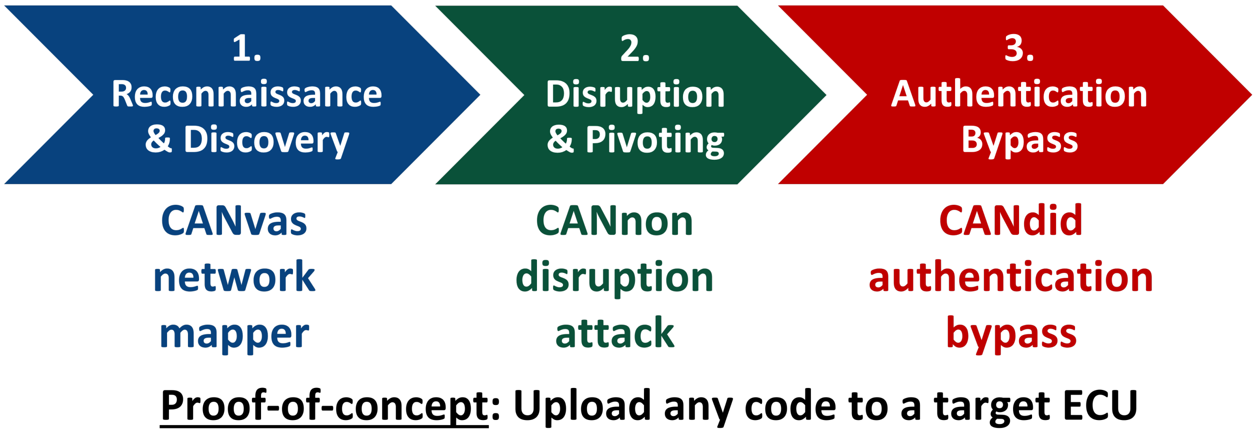

1. Reconnaissance & Discovery
2. Disruption & Pivoting
3. Authentication Bypass
CANvas network mapper
CANnon disruption attack
CANdid authentication bypass
Proof-of-concept: Upload any code to a target ECU

Figure 2: Three thesis components and final goal

ECU and succeed in their attack. There are several challenges for each stage of the kill-chain that prevent the adversary from achieving that stage's goal. However, we believe that there are disconnects between the design of the layer under attack at each stage (e.g., design of messages on the network, design of controller hardware, design of authentication protocol) and the actual implementation of that layer in a real vehicle. Thus, we construct the following as the central point in this thesis:

Thesis statement: *By identifying disconnects between design assumptions and actual implementations, we can construct an attack kill-chain that enables a remote adversary to reprogram another in-vehicle ECU and, thus, informs countermeasures in the defense of next-generation vehicles.*

For each stage of our kill-chain, we will identify a disconnect that can manifest as a vulnerability that a remote adversary can exploit. We construct attacks that exploit each vulnerability to achieve that stage's goal, and we identify key design decisions that enable each exploit. Using this information, we can inform countermeasures by providing potential defenses that can either detect or prevent such an attack in future vehicles. In what follows, we review the key contributions of this thesis and highlight the key points of each contribution, namely the disconnect we

and out of our scope. For our three kill-chain stages, we identify vulnerabilities that exist across makes and models to emphasize the impact of this thesis.

exploit, how we exploit it using a remote attacker, and then potential countermeasures. We also highlight how each stage aids the next and, thus, why each stage is necessary to achieve the final goal of enabling reprogramming on a safety-critical ECU.

1.4.1 Stage 1: Reconnaissance and discovery

We present *CANvas* [31], the first stage of this kill-chain for after the adversary gains remote access to the in-vehicle network, where we perform reconnaissance and determine which messages originate from which ECUs. To characterize the ECUs on a CAN bus, we argue that we need tools analogous to network mappers for traditional networks that provide an in-depth understanding of a network's structure. To this end, our goal here is to develop an automotive network mapping tool that assists in identifying a vehicle's ECUs and their communication with each other. A significant challenge in designing this tool is the broadcast nature of the CAN protocol, as network messages contain no information about their sender or recipients. To address this challenge, we design and implement *CANvas*, an automotive network mapper that identifies transmitting ECUs using just an hour of passively-collected traffic capture.

Stage goals: As detailed in Chapter 3, this stage starts with the remote adversary already in control a single (likely non-safety-critical) ECU inside a victim vehicle. Here, the adversary can gain prior knowledge about the victim vehicle by simply buying a copy of the same vehicle (i.e., same make, model, year, and trim). With this replica vehicle, the adversary will want to build a map of the vehicle's network, including the set of ECUs in the network and the messages that each ECU sends. Here, the adversary can gather information about the characteristics of the network and then use that information to plan the later stages of the attack kill-chain. When the adversary achieves the initial compromise on the victim vehicle, they will want to verify that the network matches exactly that of the replica vehicle. By confirming that both vehicle networks match, the adversary can confidently move on to the next stage of the kill-chain without worrying about new messages appearing (or expected messages disappearing) on the network, which would interfere with any attack configurations that the adversary selected for the kill-chain.

Disconnect: Following a metric initially identified by prior work on building an intrusion detection system (IDS) for a vehicle network [15], we identify a disconnect that reveals some information about the source of a given message. The authors of this IDS found that the periodic nature of CAN messages inadvertently permits any device snooping on the network to extract a timing characteristic called clock skew from each message [15]. This work tracked this clock skew and looked for changes during an adversary's attempt to inject malicious CAN traffic. While we initially tried to repurpose their approach to uniquely identify the messages that originate from each ECU, their approach could not identify messages with differing periods that originate from the same ECU and thus was not a sufficient approach for our purposes. As a result, we propose a new technique that measures clock offset (instead of clock skew) and demonstrate how this approach can uniquely identify the source ECU for any given periodic CAN message.

Results summary: Using *CANvas*, we can identify transmitting ECUs with a pairwise clock offset tracking algorithm and identify receiving ECUs with a forced ECU isolation technique. *CANvas* generates network maps in under an hour that identify a previously unknown ECU in a 2009 Toyota Prius. Here, we imagine an adversary who purchased a replica 2009 Toyota Prius in preparation for their attack. Using the *CANvas* network mapper, our adversary could now identify this unknown ECU in this particular ECU. Since this ECU produced new transmissions on the CAN bus that could affect the timing of other messages, the adversary could decide to move on to another 2009 Toyota Prius as their planned attack configurations may no longer work. From these findings, we also suggest countermeasures to prevent mapping capabilities by either removing the periodic nature of CAN messages or by changing the message ID to make it more challenging to track clock offset.

1.4.2 Stage 2: Disruption and pivoting

We present *CANnon* [32], the second stage of the kill-chain, which involves using the initial compromised ECU to disrupt transmissions from other ECUs. To achieve this disruption, we introduce a new class of attacks that leverage the peripheral clock gating feature in modern automotive microcontroller units (MCUs). By using this capability, a remote adversary with purely software control can reliably “freeze” the output of a compromised ECU to insert arbitrary bits at any time instance. Utilizing on this insight, we produce error patterns indistinguishable from natural errors and do not require message insertion. Using *CANvas*, we can identify a target ECU, select an optimal target message, and disrupt how that ECU interacts with the CAN bus. As a demonstration of this method’s capabilities, we show how *CANnon* can shut down a specific ECU without detection by modern defenses.

Stage goals: As detailed in Chapter 4, this stage starts with the remote adversary that wants to use the transmissions of the initial compromised ECU to impact the transmissions of other ECUs. Here, the adversary can transmit data onto the network to impact the timing and state of other ECUs, but they cannot use simple message injection as many modern defenses are tailored to detect these types of attacks. By finding a technique that can impact other ECU transmissions, the adversary can open the door to a number of techniques, such as forcing other ECUs to delay their transmissions for a small amount of time, forcing messages to appear on the bus at certain times, causing ECUs to enter different error states, etc. While these techniques can be performed by injecting messages (e.g., flood the bus temporarily with high-priority messages to block messages), these techniques can also be performed with physical access by tapping into the physical CAN signals. However, as our adversary is limited to remote-capable techniques, we must find a way to mimic such physical attacks using just software instructions. By achieving this capability, the adversary can then ensure better success for the next stage of the kill-chain.

Disconnect: While analyzing the impact of software instructions on physical-layer CAN signals, we identify a disconnect that enables a remote adversary to interrupt a compromised ECU’s transmission in the middle of a message. Typically, the CAN controller hardware is responsible for enforcing compliance to the CAN protocol (i.e., this hardware ensures that traffic on the CAN bus strictly follows the CAN protocol). This enforcement used to prevent software from sending malformed CAN traffic on the CAN bus and thus could not be exploited by a remote adversary. However, we find that modern microcontroller units (MCUs) on high-performance or networking ECUs (e.g., infotainment or telematics ECUs) implement a new feature called peripheral clock gating as a feature to reduce an ECU's power consumption. This feature grants the software on an MCU to disable the clock signal to a peripheral (including CAN), which effectively "turns off" that peripheral. This new clock gating instruction can permit an adversary to pause the CAN peripheral (basically, the CAN controller) in the middle of a transmission. As a result, we propose a new technique that exploits this instruction to impact other CAN transmissions on the bus and demonstrate how this approach can even trick victim ECUs to enter into a shutdown state.

Results summary: Using *CANnon*, we illustrate both a basic denial-of-service (DoS) attack and a targeted victim shutdown attack atop two modern automotive MCUs used in passenger vehicles: the Microchip SAM V71 MCU and the STMicro SPC58 MCU. We validate the feasibility of this attack against a 2017 Ford Focus and a 2009 Toyota Prius and achieve a shutdown in less than 2ms. Here, we imagine an adversary who wants to impact the transmissions of another ECU on the CAN bus. We demonstrate how a remote adversary can now craft physical-layer signals using just software and with such precision that we can perform repeated attacks against a victim and force it to shut down. The *CANnon* disruption attack enables the adversary to impact synchronization on the CAN bus, which proves useful in the next and final stage of our kill-chain. From these findings, we also suggest countermeasures to detect malicious clock gating by using a host-based power IDS that tracks changes in power consumption. We also provide countermeasures to prevent this attack by clearing transmit buffers when the clock signal is disabled or by removing clock gating altogether for the CAN peripheral.

1.4.3 Stage 3: Authentication bypass

We present *CANdid*, the third stage of the kill-chain, where we bypass authentication on another (likely safety-critical) ECU in the vehicle. Modern ECUs implement existing built-in protocols that permit authorized testers to reprogram an ECU's software if they pass authentication. If the adversary can successfully mimic this authentication, the adversary can access higher-privilege commands. With the knowledge gained from mapping the network with *CANvas* and the *CANnon* technique to disrupt ECU transmissions, we propose that a remote adversary can sniff on the network when a valid tester is connected and simply observe and capture a single valid authentication. This authentication contains a random challenge from the target ECU along with the correct response from the valid tester. With this challenge-response pair, the adversary will force the target ECU to send the same challenge, which permits the attacker to replay the previously-observed response and gain access to code upload commands. Once our attacker successfully gains this reprogramming access, they can then deploy their final attack payload. For our work, we demonstrate this ability by simply uploading a prior version of code.

Stage goals: As detailed in Chapter 5, this stage starts with the remote adversary that is simply sniffing on the CAN bus, waiting for the vehicle to be taken to a technician that updates the software on a target safety-critical ECU. Here, the adversary can simply observe and capture a valid challenge-response authentication. With this valid challenge-response pair in hand, the adversary will then need to find a method to launch a replay attack. Considering that modern authentication protocols may use unique keys for each instance of a vehicle, it is not sufficient for an adversary to reverse-engineering the authentication protocol and simply use that protocol during the attack. By finding a technique where the adversary can force the target ECU to produce the same challenge as seen in the previously-captured valid authentication, the adversary simply needs to replay the captured response. However, as the challenge is random, the adversary must identify a method to exhibit some control over the challenge's source of randomness. This control must fine-grained enough to enable a sufficiently high likelihood of reproducing the observed challenge. Defense technologies can permit a tester to connect to the network but only permits a handful of attempts before locking out the authentication. As a result, our adversary must force a given challenge to reproduce at a high enough likelihood (e.g., as observed in real ECU designs, a likelihood over 20% is sufficient if only five attempts are permitted) to avoid being locked out, which would alert to an adversary's presence.

Disconnect: When analyzing the available diagnostic commands to the remote adversary, we identify a disconnect that enables the adversary to exhibit control over an ECU's source of randomness. As the challenge-response authentication protocol must seed the challenge with some source of randomness, we find that modern ECUs use processor uptime, or time since last hard reset or time since last power cycling. At first glance, processor uptime might serve as a sufficient randomness source, but an adversary should not be permitted to impact this source. Typically, since the adversary cannot know when an ECU is powered-on, the challenge in the authentication protocol should be random and prevent an adversary from launching a capture-replay attack. However, we find that ECUs permit an adversary to call a diagnostic command that performs a hard reset, thus granting the adversary control of when processor uptime resets. As a result, we propose a new technique that exploits this command to control the source of randomness (and thus the challenge) and then demonstrate how this approach enables an adversary to force an ECU to produce a specific challenge, enabling a replay attack.

Results summary: Using *CANdid*, we can select any given challenge on two real ECUs and force a challenge to repeat up to 25% likelihood. We also demonstrate this attack on a gateway ECU that uses a 16-byte key and achieve a likelihood of 12.5%, and we demonstrate that this attack works on a real vehicle. With this technique, we ultimately show that a remote adversary can authenticate and reprogram an ECU with an older version of the ECU's software. Once successful with the previous stages, this work will enable our attacker to download attack code of their choosing to the target. While there are many methods of using manipulated code to launch the final attack, we provide a proof-of-concept by successfully uploading a previous version of code to the target ECU. If an adversary can perform this attack kill-chain even with modern network defenses in place and across multiple makes and models, then we would have found novel key vulnerabilities that will require countermeasures in future network designs.

1.5 Outline

We organize the rest of this thesis into the following chapters. Chapter 2 details the relevant background for this thesis and expands on taxonomy of prior work at each stage of the attack kill-chain. In Chapters 3, 4, and 5, we detail the three key components of this dissertation: (1) *CANvas* enables a remote adversary to identify network participants and their transmissions even on a broadcast network, (2) *CANnon* enables a remote adversary to disrupt the transmissions of another ECU without using CAN-formatted messages, and (3) *CANdid* enables a remote adversary to authenticate with another ECU without access to any secret keys or algorithms. With each kill-chain stage, we identify a disconnect between design and implementation that manifests as a vulnerability for a remote adversary to exploit. We discuss how such an adversary can perform each kill-chain stage while deceiving modern defenses. We also detail countermeasures that can either detect or prevent these attacks with consideration to the demands of the automotive industry. Then, in Chapter 6, we demonstrate how our kill-chain stages enable a proof-of-concept attack demonstration against a real powertrain ECU. Finally, Chapter 7 details the impact of this research on other related fields, reflects on the implications of this research with a focus on how to identify these elusive vulnerabilities, and concludes with future work.

2 Background and Prior Work

In this chapter, we discuss relevant background on a network protocol widely used in modern vehicles. We discuss the multiple layers involved in the CAN protocol [33], and we discuss the diagnostic protocol that operates over a vehicle's CAN bus. Then, we discuss the remote adversary model that we follow in this thesis, and we cover the capabilities and limitations of our adversary. Given the necessary background and adversary model, we provide an overview of prior work on attacks, how these attacks transitioned from requiring physical access to attacks that can be performed remotely, and then how modern defenses have adapted to these known remote attack techniques. We also discuss how attacks and defenses in the vehicle space are closely related to other settings.

2.1 Relevant background

We now detail the relevant protocols found in modern vehicles and the layers that a remote adversary will need to navigate for their attack kill-chain.

2.1.1 Inside an ECU

Modern vehicles contain tens of Electronic Control Units (ECUs) that control a number of subsystems, ranging from safety-critical (e.g., engine control, brake control) to non-safety-critical systems (e.g., infotainment, telematics). Each ECU typically has a microcontroller unit (MCU) that interacts with hardware within the ECU and operates the necessary software for that ECU's function (e.g., engine control, brake control). The software on an ECU's MCU reads inputs from sensors (e.g., wheel speed), writes outputs to actuators (e.g., fuel pump), and communicates with other ECUs. To communicate with other ECUs, most vehicles employ the Controller Area Network (CAN) protocol. The CAN protocol stack as shown in Figure 3 is composed of the application layer, data link layer, and the physical layer. The functionality of an ECU (e.g., engine control, driver assistance) is described via high-level software running at the application layer. For actu-

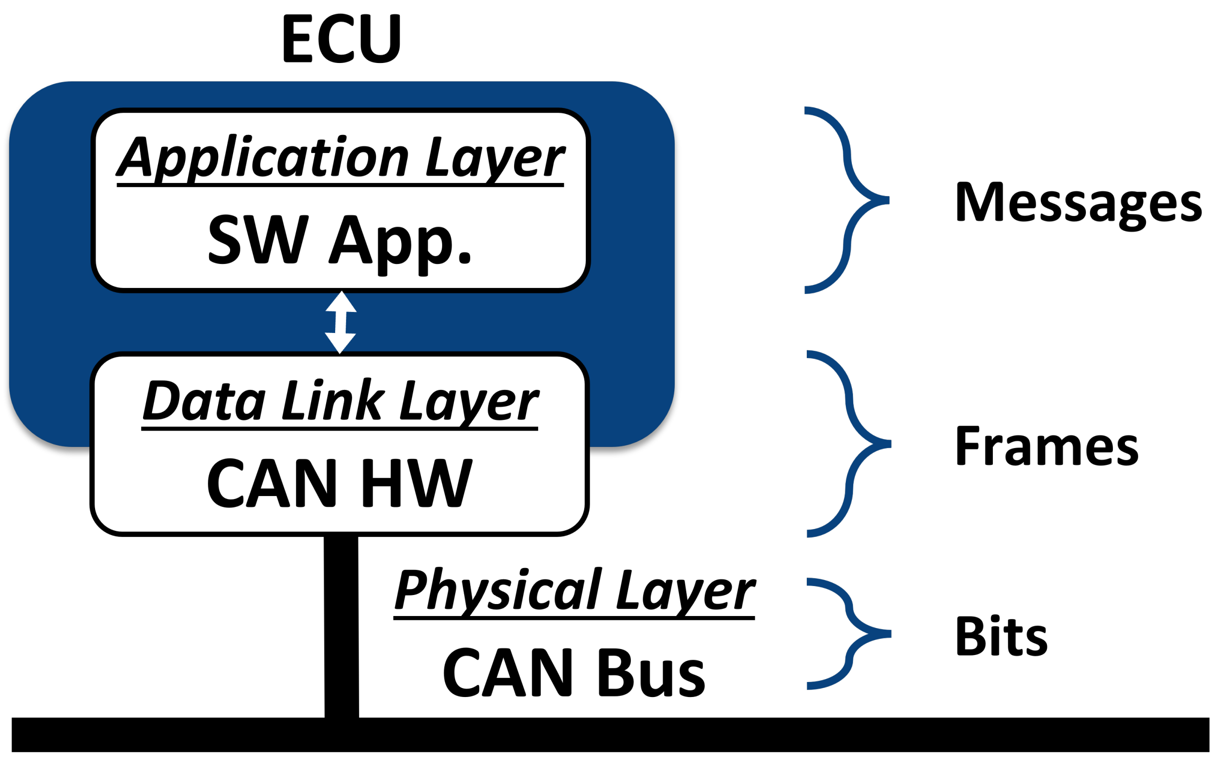
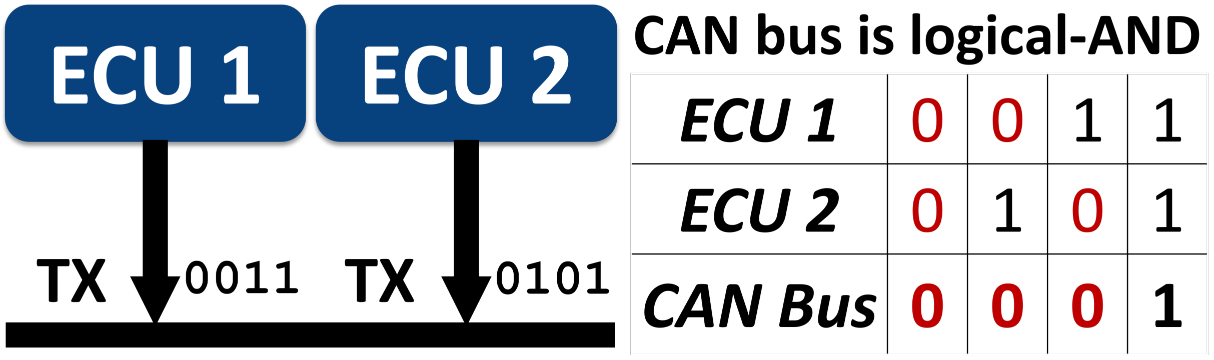


Figure 3: ECU communication stack

ation and sensing functionality, messages are transmitted and received by the application layer through the lower layers of the communication stack. To send data to another ECU, the application layer creates a CAN message with a priority tag (also referred to as message or arbitration ID) and its payload. The application transfers this message to the CAN data link layer, where various control and integrity fields are appended to generate a *frame*, which is transmitted serially via the CAN physical layer. To receive a message, a recipient ECU's data link layer interprets and validates the CAN frame prior to delivery of the message (ID and payload) to the application layer.

2.1.2 CAN physical layer

The physical layer of the stack (i.e., the physical CAN bus) consists of a broadcast communication medium between multiple ECUs. The bus has two logical states: the dominant (logical-0) state, where the bus is driven by a voltage from the transmitting ECU, and the recessive (logical-1)



ECU 1	0	0	1	1
ECU 2	0	1	0	1
CAN Bus	**0**	**0**	**0**	**1**

Figure 4: CAN physical layer

state, where the bus is passively set. The effective bus state is the logical-AND of all transmitting ECUs' outputs as illustrated in Figure 4. ECUs connected to the CAN bus communicate at a pre-determined bus speed set by design based on the physical limitations of the bus. The length of each bit is directly determined by the set speed. For example, an ECU communicating at 500Kbps transmits the dominant signal for $2\mu s$ to assert a logical-0. Similar to other asynchronous protocols (e.g., Ethernet), CAN nodes rely on frame delimiters for interpreting the start and stop of CAN frames. Each ECU (re)synchronizes its internal clock based on observed transitions on the bus.

2.1.3 CAN data link layer

Each CAN message from an ECU uses its assigned message ID (interchangeably referred to as the ID or the arbitration ID), which determines its priority on the CAN bus and may serve as an identifier for the message's contents. These messages are transmitted and received at the physical layer by an ECU's CAN controller as CAN data frames in the format depicted in Figure 5. All ECUs in the network with a queued message simultaneously start to transmit their message at the same time. During the arbitration ID field, all but one ECU will eventually stop transmitting based on CAN's arbitration resolution. CAN is designed to support collision detection and bit-wise arbitration on message priority to allow higher-priority messages to dominate the network. The arbitration of these messages is performed on the message ID field of a data frame, where a lower

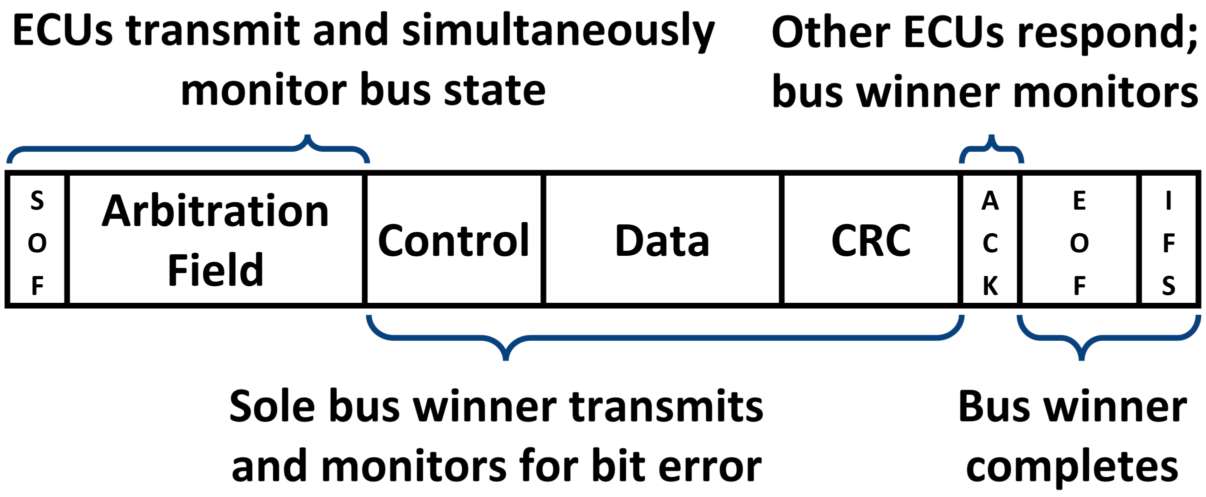

ECUs transmit and simultaneously monitor bus state
Other ECUs respond; bus winner monitors
SOF
Arbitration Field
Control
Data
CRC
ACK
EOF
IFS
Sole bus winner transmits and monitors for bit error
Bus winner completes

Figure 5: CAN frame format

ID indicates a higher priority. This priority-based arbitration process sets a 0-bit as dominant and a 1-bit as recessive, following the logical-AND bus as seen in Figure 4. Since a 0-bit is dominant, a message with a lower ID will get priority on the CAN bus and will be sent before a message with a higher ID that is queued at the same time.

The CAN data frame illustrated in Figure 5 has four logical sections that we further detail here: (1) arbitration, (2) data transmission, (3) acknowledgement (ACK), and (4) end-of-frame (EOF) and inter-frame spacing (IFS). Upon detection of an idle bus, an ECU initiates the frame transmission with a dominant start-of-frame (SOF) bit followed by the arbitration ID. Due to CAN's asynchronous nature, multiple ECUs may begin transmission at the same time. While transmitting the ID, an ECU monitors the bus state and stops transmitting if it observes a bit different from the one transmitted. A received dominant bit during a recessive transmission by a node indicates the transmission of a higher-priority message by a different ECU. By the end of arbitration, a single ECU with the highest-priority frame wins access to the bus and continues transmitting. The bus winner transmits the rest of its frame and, for each transmitted bit, monitors that the bus state matches the transmitted bit. During the ACK slot, the transmitter asserts a recessive bit while all receiving ECUs transmit a dominant bit to indicate correct reception. Finally, the sender transmits recessive EOF and IFS bits, where the IFS is the minimum space between two frames on the bus. After the IFS, the bus is idle and holds a recessive state until the next transmission. Each ECU can transmit multiple IDs, but each ID *should* only originate from a single ECU.

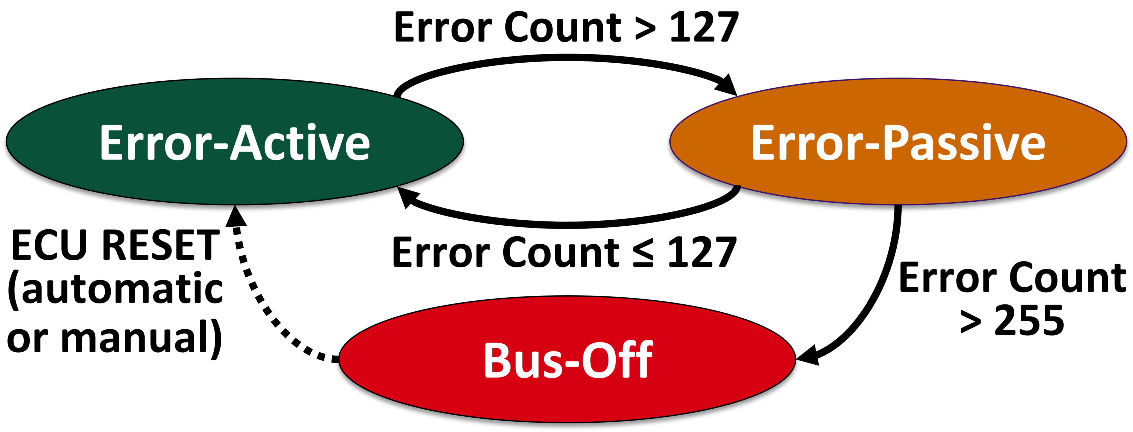


Figure 6: Error-handling states

Error handling and bus-off state: Error handling is an essential feature of the CAN protocol, providing robustness in automotive environments. The CAN protocol defines several types of errors; we detail two relevant error types, namely the *bit error* and *stuff error*. A bit error occurs when the transmitting node detects a mismatch between a transmitted bit and the bus state (outside of the arbitration and ACK fields). A stuff error occurs in the absence of a stuff bit, which is a bit of opposite polarity intentionally added after every five consecutive bits of the same polarity. When an ECU detects an error, it transmits a 6-bit error flag on the bus that can destroy the contents of the current frame. Depending on the error state of the ECU, the flag may be a sequence of recessive or dominant bits. Each ECU maintains error counters that are incremented upon a transmission error[2] detection and decremented upon a successful transmission. As depicted in Figure 6, there are three error states based on the error count: (1) error-active, (2) error-passive, and (3) bus-off. An ECU in error-active state represents a "low" error count and transmits a 6-bit *active* (dominant) error flag; an ECU in error-passive indicates a "high" error count and transmits a 6-bit *passive* (recessive) error flag. If enough errors are detected

[2]There is a separate count for reception errors, but it is not relevant to this work. All references to error count refer to the transmission error count.

and the count surpasses 255, then an ECU transitions to bus-off, where it will shut down its CAN operations and effectively remove itself from the bus.

2.1.4 CAN application layer

All vehicles produced for the U.S. market in 2008 and after are required to implement the CAN protocol for diagnostics purposes [34]. Many vehicles will often employ either one, two or three CAN buses. In the event of three CAN buses, it is likely that the vehicle has one bus for powertrain components (engine, transmission, etc.), one bus for infotainment components (radio, etc.), and another for body components (door controller, headlights, etc.). These CAN buses are usually exposed through a vehicle's On-Board Diagnostics (OBD-II) port, which is typically located under the steering wheel. This port is the typical access point for attacks that require physical access [7].

The CAN protocol is defined as a message broadcast bus, which means that ECUs are connected to a shared network where all ECUs can receive all transmissions [33]. Due to the nature of this broadcast bus, it is not possible to send a message to a specific ECU where only that ECU has access to the message. In the CAN protocol, after a message is broadcast to the network, devices that correctly receive (i.e., all bits received at physical-layer, error-correction checks pass, and frame is properly formatted) this message will acknowledge their reception. A typical CAN setup for a vehicle will grant each ECU with a unique set of IDs, and each message will be labeled with an ID that is then transmitted onto the bus. An ECU will be responsible for a subset of the message IDs seen in the network, and each message ID will only be sent by a single ECU. Each message is queued by a software task, process or interrupt handler on the ECU, and each ECU will queue a message when the message's associated event occurs. This message ID serves primarily serves as a label for the data contained in the CAN message's payload. For the standard CAN protocol, the range of the 11-bit message ID can be split into two groups: 0x000-0x6FF for regular CAN traffic and 0x700-0x7FF for diagnostic messages. Any device that is on a CAN bus can claim any message ID, which enables a compromised node to mimic other ECUs or even a diagnostic tester.

2.1.5 UDS session and application layers

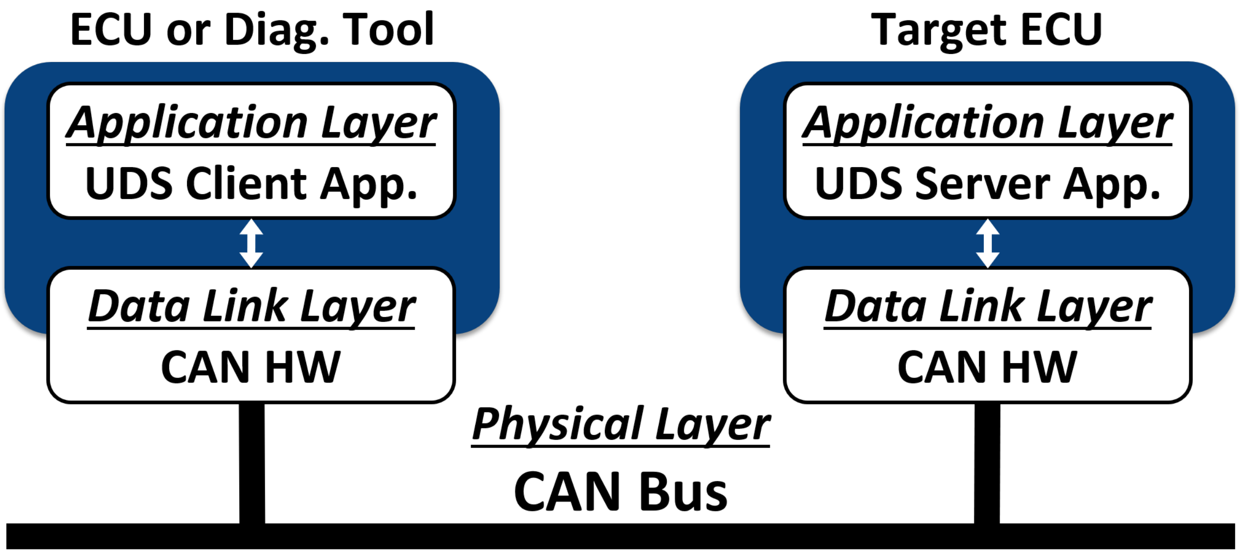


Figure 7: UDS application layer above CAN bus layers

With the CAN protocol serving as the lower layers of in-vehicle communication, we now detail the Unified Diagnostic Services (UDS) protocol, which defines a session and application layer above CAN as depicted in Figure 7. UDS serves as an interface for running vehicle diagnostics, debugging, and configuration on in-vehicle ECUs. There are typically only two participants to any UDS communication: a *client* that is typically the diagnostic tester physically connected to the vehicle's OBD port and a *server* that is a UDS-enabled ECU on the vehicle's CAN bus. Each UDS server runs a server application that is assigned a server ID (i.e., just a unique CAN message ID within the range of 0x700-0x7FF) to identify CAN messages originating from the UDS server. To communicate with a UDS server, the diagnostic tester (or a compromised ECU) must use the appropriate UDS client ID (i.e., another unique CAN message ID) for the given server.

The UDS session layer defines various session types between a server and client, where a given type defines the available services accessible to the client. A UDS server will initially start in the default session state, and the client can request a different session type without any additional security check. These session types are ultimately defined by the OEM but typically include, at a minimum, the default and programming session types. In the default session, the client is limited to a few services, such as requesting Diagnostic Trouble Codes (DTCs). We now detail some UDS services relevant to this thesis:

- *DiagnosticSessionControl*: A UDS client can use this service to request a UDS server to enter a specific session type, where the type determines the available services to the client. For example, the "programming" session can enable services to download firmware and perform other critical read/write operations while the "extended diagnostic" session can enable services to reconfigure sensors and actuators.
- *ECUReset*: A UDS client can use this service to request a UDS server to reset itself following one of several reset types. These reset types typically include a "hard" reset that simulates power cycling and a "soft" reset that re-initializes some of the ECU's firmware. These resets do not require any authentication and can be requested at any time.
- *SecurityAccess*: A UDS client can use this service to request access to privileged services on a UDS server but only *after* the server enters the programming session. The ECU designer defines a variety of security levels that grant fine-grained control over the set of commands that become available to an authenticated client. We find that the highest security level (i.e., level 1) typically grants the ability to flash an ECU's software while lower security levels enable the modification of ECU configuration files.
- *RequestDownload*: A UDS client can use this service to initiate a download transfer to the UDS server but only *after* the client authenticates with a UDS server via SecurityAccess. With the proper security level (i.e., level 1), the client can then download new firmware to the server using the UDS TransferData service.

As mentioned above, the SecurityAccess service enables a client to access privileged commands, such as downloading code to an ECU, on a server in the programming session. The set of available commands depends on the chosen security level, and the client must pass a challenge-

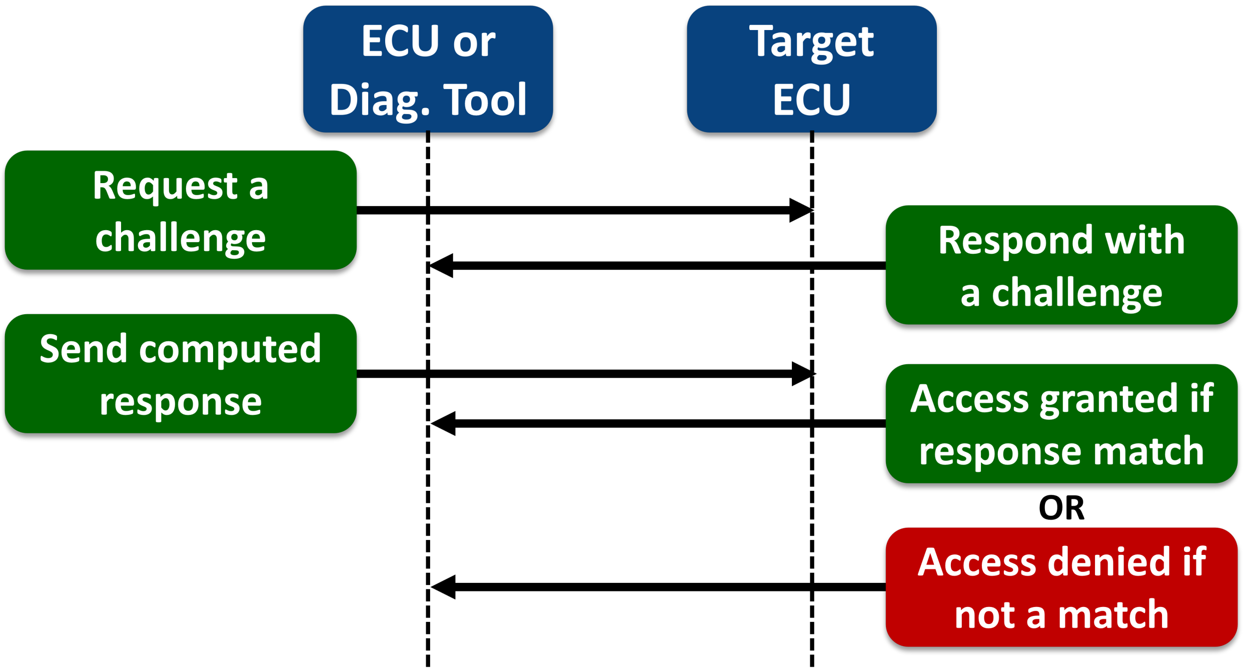


Figure 8: UDS SecurityAccess service

response authentication for that given security level (i.e., the secret key and encryption algorithm can differ per level). As manufacturers expect that only authorized service technicians should reprogram ECUs, getting access to higher-privileged commands *should* require passing authentication with proprietary diagnostic tools. As only an authorized user should have these shared secrets, this service uses challenge-response authentication to confirm that the user knows these secrets. As depicted in Figure 8, there are four main steps to UDS SecurityAccess:

1. After the UDS server enters the programming session, the UDS client initiates SecurityAccess with a request to the server. This request message also specifies the requested security level (in this case, level 1 to access reprogramming capabilities).

2. After the server receives this request, it computes a challenge C using a pre-defined method in the server's UDS application. This challenge C is then sent to the requesting client.

3. The client receives the challenge C and then computes the response R_{client} using the shared (at build time) encryption algorithm. This algorithm is pre-defined in both UDS applications for the client and server, and the algorithm takes in two inputs: the challenge C and a

shared (at build time) symmetric key $K_{server,client}$. After computing the response R_{client}, the client then sends it to the server.

4. With the client's response R_{client} in hand, the UDS server computes the expected response R_{server} by following the same encryption algorithm as the client, where the expected response R_{server} is a function of challenge C and key $K_{server,client}$. Then, this response R_{server} is compared against R_{client}, to determine if they match. If they match, the server responds positively (thus granting access); otherwise, the server responds negatively with an error code indicating an invalid response.

If the client successfully authenticates via UDS SecurityAccess with the highest security level (i.e., level 1), then the client can request a download and transfer data to the server via the RequestDownload and TransferData service, respectively.

2.2 Our remote adversary model

In this thesis, we consider an adversary who has compromised the software of a single in-vehicle ECU, which is likely to be an infotainment or telematics ECUs [9]. Several prior and recent works demonstrate the real existence of vulnerabilities to both remotely compromise in-vehicle ECUs and gain the ability to take control of physical vehicle functions via CAN transmissions [10, 22, 35, 36, 37]. These works also demonstrate that remote attacks can occur at a large scale since a single vulnerability can be present across hundreds of thousands of vehicles [10]. In addition to this, as market research estimates that about 150 to 250 million connected cars were on the road in 2020 [26, 27, 28], the number of vehicles with remote interfaces via an infotainment or telematics ECU will likely continue to increase. As a result, it is clear that the attack vectors for a remote adversary will become more prevalent and pose a significant threat to vehicle security.

Following the same assumptions as prior work, we assume that such a remote adversary can gain complete control of an ECU and subsequently modify the ECU's software, including application software running on the ECU and any application programming interfaces (APIs) available to this software [30, 17, 15, 16]. The primary limitation here is that all adversarial capabilities are limited to *only* software manipulation. Our model does not allow for any direct physical modifications or probing to any of the vehicle's components. We want to design an attack kill-chain that can be simultaneously performed on thousands of vehicles without requiring any hands-on access to the vehicle. We also assume that the vehicle is equipped with state-of-the-art defenses, which can detect attacks that cause the network traffic to sufficiently deviate from normal operations. For example, attacks that inject malicious CAN traffic that typically come from in-vehicle ECUs would easily be detected. However, as a technician who connects to the vehicle will be a new device communicating new traffic to a CAN bus, diagnostic commands over the bus are permitted as long as any attempt limits are not breached (e.g., systems that permits only five attempts to authenticate). On top of this, we also assume that the victim vehicle follows a realistic hardware design so we base our investigation on real ECUs and automotive-grade hardware. Finally, another limitation that we address throughout the thesis is the need for all attacks to be robust. We make the assumption that any attacks that cause collateral damage to other ECUs and not the intended ECU can be detected by defenses.

While there are a number of limitations against the adversary, there is one major advantage that a remote adversary can exploit: the fact that vehicles are essentially replicas of each other. Here, we envision an adversary who simply purchases or obtains a copy of the victim vehicle that has the same make, model, year, and trim. By building an attack kill-chain for this vehicle, the adversary can potentially attack thousands so the cost of a vehicle here is reasonable [9]. After identifying a method to remotely compromise an in-vehicle ECU, the adversary can gain more information from this vehicle's network to launch our kill-chain. They can access components physically for testing purposes, inject malicious traffic without fear of repercussions, and even trigger alerts from a vehicle's defenses. We assume that the adversary can block outgoing alerts to the vehicle manufacturer, and we assume that a vehicle's network will not be permanently locked down in the event of a detected attack. As we progress through each stage of the kill-chain, we identify what work can be performed on a replica vehicle and also how the adversary will launch the attack once in the victim vehicle's network.

Our model is realistic: As outlined in two U.S. agency reports, a remote adversary is considered the highest risk factor for the automotive community and passenger safety and thus taken seriously by the industry [29, 38]. Security efforts by vehicle manufacturers (e.g., introduction of IDSes) place significant focus on defending after such an adversary breaches the network as this access opens the door to an adversary who can simply inject messages onto the CAN bus [29]. In short, if an ECU designer can write a line of code to take some action on a given ECU, then a remote adversary who compromised that ECU can run the same line of code in our threat model.

2.3 Prior work for each stage

Table 1: Taxonomy of prior work for each kill-chain stage

Kill-chain stage	**Physical approaches**	**Remote approaches**	**Defenses**
Reconnaissance	[7, 39, 40]	[41, 10, 42, 15, 43, 18]	[44, 45, 15, 18]
Disruption	[46, 47, 31]	[30, 41, 10, 42]	[43, 15, 48, 49, 20, 16, 17, 50]
Authentication bypass	[7, 51]	[10, 22, 52]	[52, 53]

Each of our attack kill-chain stages has followed a general evolution that we now explore. After the initial demonstration of in-vehicle network attacks by Koscher et al. [7], there were several demonstrations of other attacks that also required physical access. However, as these physical attacks were considered impractical and especially after the real remote attack by Miller et al. [10], we started to see attacks that could be performed by a remote adversary. As a response to these remote-capable attacks, we then started to see modern defenses that focused on these new remote techniques. In what follows, we detail how the taxonomy of prior work has progressed following this trend for each kill-chain stage.

2.3.1 Reconnaissance and discovery

With control over a vehicle's infotainment ECU, our remote adversary must first identify the ECUs in the network and characterize their communications on the CAN bus. Obtaining information about CAN bus participants and building a map of their communication is a challenging task. Unlike other protocols (e.g., Internet Protocol (IP)) that contain source and destination addresses, the CAN protocol does not include such information. As detailed in the above background, the message ID simply acts as a label for the data in a given CAN message and gives no information on the actual source ECU. Knowing which ECU sends a given message ID can prove useful for a variety of reasons, such as identifying proprietary message content [54], analyzing individual ECUs [7], and confirming ECU configurations post-update [31]. In the context of our adversary, such a technique would be useful for surveying a number of vehicles with the goal of picking a target make and model vehicle, which Miller et al. had to do [9] prior to their well-known attack on a 2014 Jeep Cherokee [10]. Then, by mapping a replica of the victim vehicle, the remote adversary can compare the network map of the actual victim vehicle to ensure that it matches the expected network map.

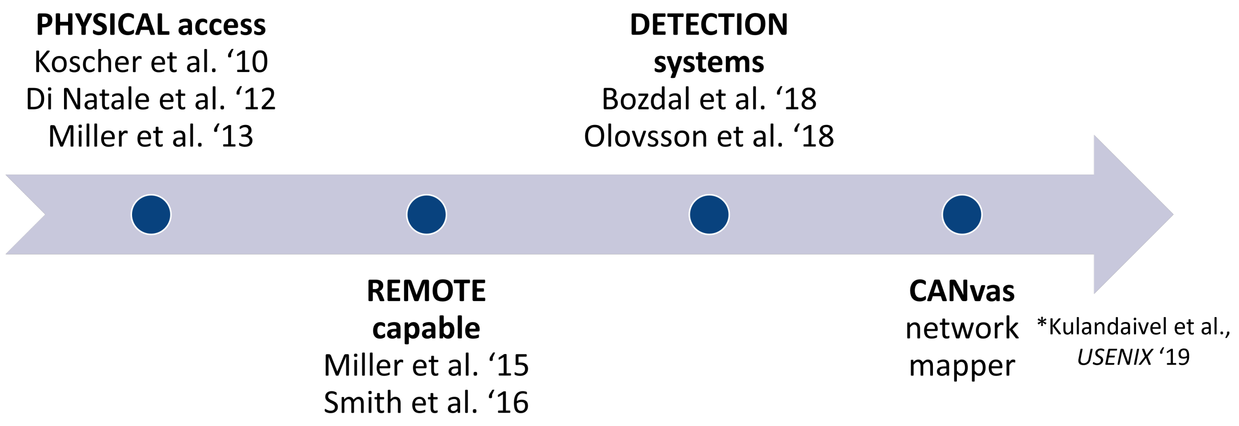


Figure 9: Taxonomy of prior work in reconnaissance and discovery

Physical access attacks: The legacy approach to mapping a CAN bus was "old-fashioned" and required physical access or could only identify a handful of ECUs [55]. One approach to map a CAN bus required physically disconnecting ECUs and looking for missing messages. Extracting the necessary information to map a CAN bus with this approach requires an unreasonable amount of effort. In a work by Koscher et al., the authors analyzed the security of a vehicle's components by manually extracting ECUs to isolate and interact with them [7]. This type of analysis requires significant time and effort or access to limited or proprietary information [9]. In addition, obtaining vehicles for extended time and with permission to disassemble is costly and expensive. Considering new model years and an increase in software updates, the frequency of analyzing an intra-vehicular network will quickly increase in time and cost requirements. Prior work has also identified message sources for the purpose of identifying a compromised ECU [18, 40], but these efforts either require several hours of data and would not be practical for a remote adversary.

Remote capable attacks: One major limitation with the legacy approach to mapping a CAN bus is the need for physical access to get a complete map of a given vehicle. To address the challenge, we could consider using techniques that a remote adversary could use to build a realistic mapping solution that identify all ECUs and the set of messages they send/receive. One approach involves injecting diagnostic commands to temporarily shut down an ECU. These diagnostic messages were originally intended for a mechanic to perform ECU testing but can be exploited to pause an ECU's communication [41, 10, 42], where the adversary can simply observe what messages no longer appear on the bus. A limitation with this approach is that these commands are not always available on all ECUs, and it is critical for our adversary to map *every* ECU in the vehicle as an unexpected ECU could negatively impact attack configurations and cause an adversary to have collateral damage, which may open the door to detection. Following initial efforts on fingerprinting using timestamps collected by an ECU's software [39, 40], other work has improved on their basic approach by identifying potential pitfalls [43, 15, 18]. In these works, the authors have demonstrated that clock characteristics can aid in identifying the source of messages even when on a broadcast network. However, as we discuss in Chapter 3, these approaches cannot successfully be repurposed to accurately map a real vehicle.

Defenses versus remote adversary: Our adversary must consider intrusion detection systems (IDSes) that can detect a large number of malicious message injections [44]. To use diagnostic messages to temporarily pause an ECU's transmissions, our adversary would need to inject these messages for each individual ECU in a victim vehicle. Here, the adversary would need to map a replica of the target make and model and also map the victim vehicle instance to confirm that the network maps match. Injecting a large number of diagnostic commands (recall that modern vehicles contain an increasing number of ECUs [26]) can expose the adversary to detection. Likewise, even if the adversary only needed to inject a small number of messages, this traffic should not be wasted on the reconnaissance stage. Also, authentication for CAN devices could implicitly prevent mapping a vehicle. Prior work, such as the TCAN system [45], requires the addition of a new device, access to two locations on the bus, and a static authentication table labeled with each ECU's timing characteristics. These characteristics can and will change due to clock drift so techniques that rely on static timing characteristics [15, 18] would not succeed here.

2.3.2 Disruption and pivoting

After mapping the victim make and model and then confirming those findings on the victim vehicle instance, our remote adversary must then construct a method to maliciously disrupt other ECUs with the intent of impacting their communications. With such a capability, our adversary could aid the following kill-chain stages, especially if those techniques requiring impacting CAN bus transmissions to the adversary's advantage. Here, we focus on works that aim to shut down ECUs; while this is not the ultimate aim of our adversary, shutting down ECUs (and the methods to achieve this) can open the door to related techniques that can be exploited. While a large error count in a non-adversarial scenario is indicative of a faulty node and, hence, isolation (or even shutdown) is a logical solution to prevent disruption of the whole network, an adversary can misuse the error mechanism by causing intentional errors, forcing an ECU to transition into the bus-off state and thus causing the ECU to shut down CAN communication. However, producing intentional errors on the CAN bus without direct access to the physical medium is challenging. One reason is the compliance to the CAN protocol enforced by hardware CAN controllers designed and certified for robustness. Thus, without access to the physical medium, an adversary can only control the ID and payload but *not* the transmitted frame. Nevertheless, prior work has demonstrated limited success in operating under these constraints to cause a shutdown.

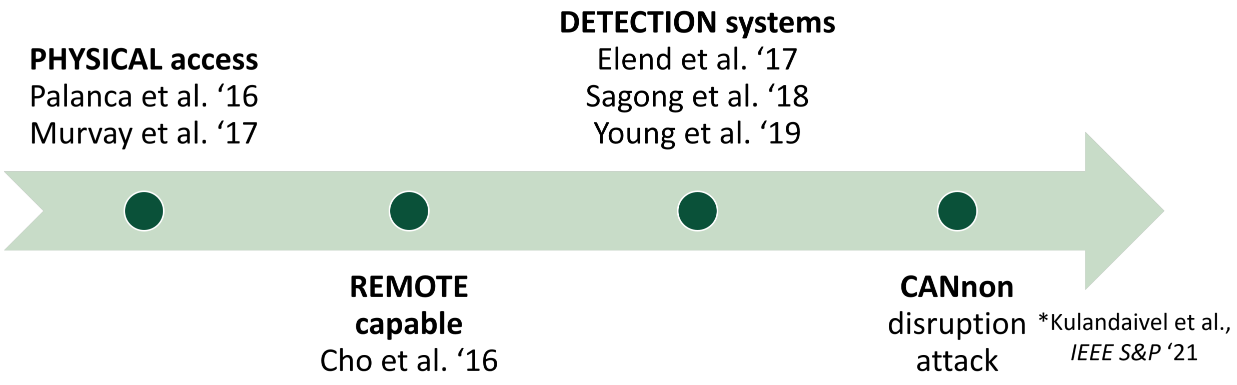


Figure 10: Taxonomy of prior work in disruption and pivoting

Physical access attacks: The legacy approach for shutting down an ECU and thus disrupting its communication required physical access. An adversary could easily bypass the CAN data link layer and inject bits by either sending signals directly to the physical bus or modifying the CAN controller to disobey the protocol. An adversary can also use this access to directly inject dominant bits at any time during a victim's transmission and cause bit errors. Several works use this approach to demonstrate effective shutdown attacks that are difficult to detect as such errors are indistinguishable from genuine bus faults [46, 47, 31]. These attacks have real-time feedback from the bus, enabling a reliable method of shutdown. However, because they require physical access, they are considered impractical both in research and practice as there are easier alternatives to cause harm [29].

Remote capable attacks: In contrast to the legacy approach, prior work only exploits software and demonstrate the ability to overwrite messages and exploit CAN's error-handling mechanism without physical access [30]. The authors aim to shut down an ECU by causing an error in the target ECU. By exploiting the error-handling protocol in CAN to effectively remove an ECU from the network, they choose to increment the error counter of a target by causing a bit error. This error occurs when a transmitting ECU reads back each bit it writes; when the actual bit is different, the ECU invokes an error. Here, an adversary must estimate a victim's message transmission time. As most CAN messages *theoretically* tend to be periodic, an adversary could perform this attack via empirical analysis. Using these estimates, a remote adversary in control of the MCU's software can transmit an attack message at the same time and with the same arbitration ID as the victim. This approach results in two nodes winning control of the bus and intentionally violates the CAN bus protocol. A specially-crafted payload (a dominant bit in place of a victim's recessive bit) can cause the victim to detect a bit error and retransmit its message; by repeating this attack, the victim eventually shuts down. Recent work demonstrates that this attack is not reliable as the deviation of periodic messages varies significantly in practice [31].

Defenses versus remote adversary: As mentioned above, using an intrusion detection system (IDS) is a common approach for protecting CAN bus networks. Many IDSes operate as software applications and are limited to reading messages passed up the communication stack by CAN hardware. These run on any ECU and do not require special hardware, making them an attractive solution. For instance, they can use statistical techniques based on message timings and content [43, 15, 48, 49]. A recent U.S. agency report discusses how companies working closely with automakers have access to proprietary information on the expected content of CAN messages, enhancing their ability to create application-layer defenses [29]. Another class of IDS that makes use of this proprietary information are machine learning [56] and traffic anomaly IDSes [57], which analyze message timing and payload to detect an attack. Application-layer IDSes can detect both diagnostic shutdown attacks [41, 10, 42] and message overwrite attacks [30] as they require transmitting additional CAN frames on the bus. As such, any application-layer defenses that measure message timing or content can also detect attacks that transmit entire CAN frames or significantly disrupt valid transmitted frames.

Additionally, recent industry solutions propose secure CAN transceivers that operate at the data link layer [20]. These transceivers can *prevent* a compromised ECU from attacking a CAN bus by either: (1) invalidating frames with spoofed CAN IDs, (2) invalidating frames that are overwritten by a compromised ECU, and (3) preventing attacks that flood the bus with frame transmissions. Attacks that require physical access are outside their scope. These transceivers are a promising approach to defending against attacks requiring message injection as the transceivers would destroy any illegitimate frames based on their IDs. Another approach for IDSes is to directly access the physical layer (i.e., measuring bus voltages). These defenses detect sudden changes over a series of CAN frames (or even a single frame) by creating a profile of the expected voltages [16, 17, 50]. These works find that each ECU applies a unique voltage that is measurable across an entire CAN frame. If an illegitimate transmitter attempts to spoof a victim's message, the voltage measured across the frame could identify a potential attack. This approach can detect the message overwrite attack [30] because a single frame will start with two simultaneous transmitters followed by only the overwriting compromised ECU; a distinctive change in voltage for the rest of the frame indicates an attack.

2.3.3 Authentication bypass

After the adversary maps the victim network and disrupts individual ECU transmissions, we move to the next stage of the attack kill-chain: authentication bypass. Here, our adversary aims to get access to higher-privileged commands on the target safety-critical ECU, including commands that enable the adversary to reprogram the target ECU. With such a capability, our adversary could inject their final attack code that could avoid constraints coded into the target ECU (e.g., speed thresholds for when brakes can be disabled) and lay dormant on multiple vehicles with the intent to launch a time-triggered attack at scale. The UDS SecurityAccess service provides a method of authentication, which permits an authorized user that has successfully authenticated to get privileged access. With regard to authentication bypass attacks, several examples in prior work demonstrate success in passing authentication with an unauthorized device. If an adversary could successfully pass a challenge-response authentication with a UDS server, then this access could enable them to reprogram the server ECU (ideally, one that is safety-critical).

Physical access attacks: Authentication via challenge-response should, at a minimum, use both a random challenge and a not obvious security algorithm to make a brute-force at-

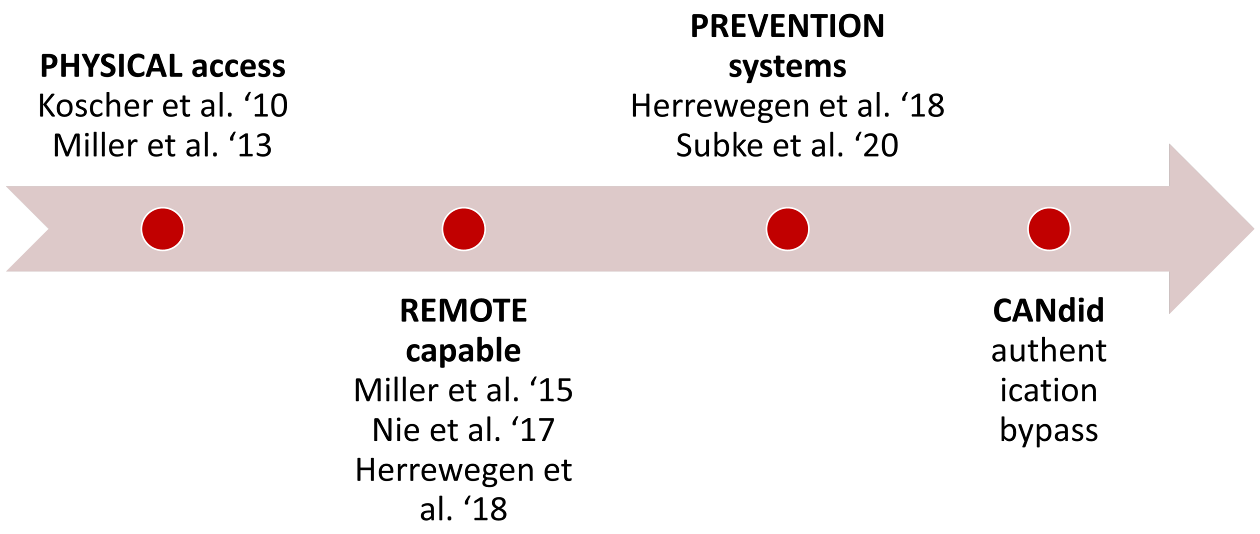


Figure 11: Taxonomy of prior work in authentication bypass

tack impractical in a realistic setting. However, researchers have found vehicles that use small responses that can be brute-forced over a few days and other vehicles with a simple security algorithm, such as simple addition, bit shift, or XOR with fixed secret [7, 51]. As UDS SecurityAccess implementations typically have a limit on the number of authentication attempts, brute-force attacks that happen in the victim vehicle can take a significant amount of time as the ECU under attack would need to be physically reset every few attempts [7], which would be impractical for a remote adversary. In addition, prior work found that some implementations of SecurityAccess did properly use a psuedorandom number generator (PRNG) to seed the challenge [10]. However, an attempt to brute-force the response would require an even longer brute-force attack and be impractical for a remote adversary.

Remote capable attacks: Challenge-response authentication also ensures that the secret key is not transmitted on the communication bus (in this case, the CAN bus) so an eavesdropper cannot extract the secret key. However, this authentication is only secure if the secret key remains out of the hands of an adversary. One approach from prior work is to extract keys directly from either the diagnostic tester tool or the target ECU itself [10, 22]. Researchers obtain a specific make and model vehicle and reverse-engineer the software for make-specific testers and target

ECUs, where they extract both secret keys and security algorithms. Even with the ability to get all keys from an ECU reverse engineering, any adversary still needs to try out all possible keys to see which one matches once inside the victim vehicle. Likewise, researchers have launched brute-force attempts at determining the secret keys by exploiting cryptographic weaknesses in the security algorithm [52]. With knowledge of both the secret key and security algorithm, an adversary can compute the appropriate response to any challenge on-the-fly. These works also demonstrate the presence of the same keys and algorithms on the same ECU type (e.g., engine) for the same make, model and trim of a vehicle, which greatly increases the scale of an adversary's attack. Thus, an adversary can reverse-engineer or brute-force the key and algorithm for a given make and model and then can launch bypass authentication on thousands of vehicles.

Defenses versus remote adversary: With respect to the growing priority of automotive security, the modern implementation of SecurityAccess has evolved to address the weaknesses of legacy designs. Vehicles (especially fleets) can implement key diversification (i.e., a unique key per ECU instance) and even perform key rotation to change keys on a repeat basis, greatly reducing an adversary's ability to target multiple vehicles [52]. Likewise, these secret keys can be made accessible only to authorized technicians[3] through the Original Equipment Manufacturer (OEM) rather than loading them onto diagnostic testers, preventing unauthorized parties from reverse-engineering these keys. Also, considering the combination of moving secret keys to the manufacturer's secure back-end server and the use of a unique secret key per ECU instance, an adversary must perform this same amount of work for each individual vehicle. Considering the presence of CAN bus IDS, a brute-force attack would be easily detected. Considering modern implementations of UDS SecurityAccess that seed the challenge with a PRNG and the use of formally verified and cryptographically sound algorithms (not easily broken by a computationally bound adversary), these methods of attack would not be so easy to perform as the pattern is no longer obvious.

[3]This practice is already common in the automotive industry for key, immobilizer and PIN codes, where authorized parties must be a member of a national registry [53].

2.4 Other related work

The concept of a remote attack kill-chain is not unique to the vehicle space. For traditional computer networks, a cyber kill-chain often involves the following stages: reconnaissance, weaponization, delivery, exploitation, installation, command and control, and action on objective [58]. Here, we find many similarities in attack techniques, such as network mappers that can identify the nodes on a network [59] and targeted disruption attacks that are intentionally hard to detect [60], as well as in defense techniques, such as building anomaly-based IDSes [61] and strong random number generation [62]. The concept of an attack kill-chain is also seen in the Internet-of-Things (IoT), where the stages can include discovery, device entry, information extraction, preparing and installing an attack page, and action on objective [63]. While the techniques for IoT attacks and defenses may be more closely related to traditional network security, many of the challenges are the same as vehicle security, including hard-coded passwords, insufficient authorization, and low compute resources. The manufacturing limitations include systems that must be highly robust while using a few resources as possible and, thus, open the door to exploits [63]. As a result, we have seen large-scale attacks on IoT [64] that share similarities to the Jeep Cherokee attack [10].

3 *CANvas*: Remote Reconnaissance of ECUs via Automotive Network Mapping

In this chapter, we discuss our contributions to the reconnaissance and discovery stage of our attack kill chain. The contributions of the thesis in this stage are:

- **Kill-chain goal:** In this stage, we aim to build a CAN bus network mapper that can identify the source of each message and the set of ECUs in a vehicle. With this information, the remote adversary can map a stock version of the victim vehicle and then compare that map to the actual victim vehicle's network.
- **Disconnect to exploit:** We find that the periodic nature of CAN messages enables an adversary to uniquely identify the source ECU. By measuring a specific timing characteristic using just the timestamps from CAN traffic, the adversary can determine the number of unique ECUs and the set of messages they transmit.
- **Potential countermeasures:** The key vulnerability insight here is that the periodicity of messages reveals information about a message's source. Thus, we propose countermeasures that "removes" this periodicity, such as intentionally adding variance to message timing and making it difficult to track messages by their ID.

3.1 Goals for this kill-chain stage

With respect to our kill-chain, the primary role of reconnaissance is ensuring that the victim network matches the network that the adversary expects. Consider the scenario where an automaker makes a particular make and model vehicle that is sold to both the general public and parties that will intentionally modify the vehicle's network. For the latter, we envision private fleets that add or remove entire ECUs (e.g., converting a hybrid to a plug-in electric [65, 66]) or vehicle tuners who reprogram ECUs to add or remove certain vehicle functions [67]. Consequently, when the adversary enters the victim system and then discovers an additional ECU or unexpected message characteristics, the adversary could detect such a modified (compared to stock) vehicle network. If the adversary were to design their attack parameters/configurations based on the stock make and model, then those values could be invalid and result in a failed attack attempt, potentially exposing the adversary. Thus, it is critical that our adversary performs reconnaissance on the victim network to verify that it is safe for the adversary to proceed; otherwise, the adversary can decide to abandon their attack and move to another vehicle instance.

Another benefit of reconnaissance is that the adversary can quickly identify a potential target make and model before they even start to build their remote attack. As an example, for the purpose of planning their well-known exploit [10], Miller et al. analyzed the in-vehicles networks of several vehicles, which revealed that the 2014 Jeep Cherokee was the "most hackable" based on its layout of ECUs [9]. Thus, it is evident that the set of ECUs and their inter-ECU communication (as well as a tool that can extract this) can determine the vulnerability of a vehicle's ECU network. Additionally, extracting knowledge about the ECUs and their messages can help inform the adversary in planning their attack as a whole. This information can aid in both targeting certain ECUs or messages *and* avoiding others to minimize collateral damage. If the adversary accidentally impacts bystanders on the network, it could potentially expose them to detection by a driver who could then safely pull over.

3.2 Stage overview and contributions

To perform reconnaissance, we argue that the adversary needs tools similar to Nmap [59], which is used to map the structure of modern IP networks. Such mapping tools have proven useful in attack scenarios, such as identifying new and unexpected servers, attesting server configurations, and inspecting firewalls by identifying available network connections. Analogously, with such a tool for scanning a vehicle's network, we could (1) discover the set of ECUs on a stock vehicle, (2) attest to the network configuration of ECUs at the time of attack, and (3) ensure that a targeted ECU can even receive messages from a compromised infotainment.

To aid in these scenarios, an ideal network mapper would require three main outputs: (1) the transmitting ECU for each unique CAN message, (2) the set of receiving ECUs for each unique CAN message, and (3) a list of all active ECUs in the vehicle. To ensure that our network mapper is practical for the adversary, we ideally want our tool to: (a) quickly analyze a vehicle's network so that the adversary can analyze multiple vehicles and then select a target make and model, and (b) accurately identify the set of ECUs and messages so that the adversary does not cause collateral damage in a later kill-chain stage.

A key challenge we face in realizing this tool in practice is the lack of source information in CAN messages. CAN messages are "contents-addressed," i.e., messages are labeled based on their data and provide no indication to the message's sender. Another significant challenge in mapping a CAN bus is the broadcast nature of the CAN protocol; we cannot tell which ECUs have received a message. A CAN message is not explicitly addressed to its recipients, but a node can indicate it has correctly received a message. We must also consider the capabilities of the remote adversary who has complete control of the infotainment ECU's software [4] and thus read/write access to the victim's CAN bus. Any write access will be closely monitored so, for now, our adversary avoids sending CAN messages that could be detected by network defenses.

For the reconnaissance stage, we build *CANvas*, a system to accurately map a vehicle's network without resorting to vehicle disassembly. Rather than require physically isolating each ECU, our key insight is to extract message information by re-purposing two observations from prior work:

- **Identifying message source**: Prior work by Cho et al. states that clock skew is a unique characteristic to a given ECU and, thus, builds an intrusion detection system (IDS) that measures this skew from the timestamps of periodic CAN messages [15]. Using this insight, we envision a mapper that computes clock skew per unique message and uses skew

[4]To identify message destination, we do require physical access to inject errors that intentionally shut down ECUs. This type of access is acceptable here since the adversary can do this on a replica of the stock vehicle, and we do offer a software-only alternative in Chapter 4 (although shutting down multiple ECUs on the victim would still be detectable). We ultimately find that every ECU we tested correctly receives all messages, but this mapper can confirm this fact on any new stock vehicle that comes to market.

to group messages from the same sender. Unfortunately, due to shortcomings of their approach in our mapping context, we instead track the clock offset of two messages over time to determine their source.

- **Identifying message destination(s)**: In another work, Cho et al. propose a denial-of-service (DoS) attack that exploits CAN's error-handling protocol to disable a target ECU [30]. Using this insight, the mapper could disable all but one ECU via this DoS attack and observe what messages are correctly received at the hardware-level by the isolated ECU. However, due to shortcomings in their method with respect to our context, we develop a method to forcefully isolate each ECU and detect which messages the ECU correctly receives despite the broadcast nature of CAN.

We implement the *CANvas* mapper on the open-source Arduino Due microcontroller with a clock speed of 84 MHz and an on-board CAN controller. We evaluate our mapper on five real vehicles (2009 Toyota Prius, 2017 Ford Focus, 2008 Ford Escape, 2010 Toyota Prius, and 2013 Ford Fiesta) and on extracted ECUs from three Ford vehicles. We show that *CANvas* accurately identifies ECUs in the network and the source and destinations of each unique CAN message in under an hour. With this speed, the remote adversary can perform a preliminary mapping of a stock vehicle prior to compromising a victim's infotainment and then can confirm these findings [5] by simply snooping on the victim's CAN bus. We also demonstrate how the *CANvas* network mapper identified that our 2009 Toyota Prius contained an unexpected ECU and was actually modified to become all-electric. This vehicle serves as a real example where an adversary would decide to move on to another target as the vehicle network did not match as expected.

In summary, we make the following contributions:

- We design an accurate message source identification algorithm that tracks a message's relative clock offset.
- We engineer a reliable message destination identification method by isolating ECUs with

[5] Again, destination mapping cannot be confirmed once on the victim vehicle, but we find that ECUs correctly receive all messages.

a forced shutdown technique.

- We build a real implementation that maps five real vehicles and extracted ECUs along with two real examples of motivating use cases for mapping.

3.3 Problem and system overview

Unlike most traditional packet-switched networks, CAN messages do not have fields that identify the message's source and destination(s), which makes the mapping problem difficult. To develop a mapper that will aid the goals of this kill-chain stage, we formulate three required outputs for *CANvas*:

ECU enumeration: The importance of enumerating ECUs is evident as it can highlight new or absent ECUs. Note that in our attack scenario, it is *not* necessary to know an ECU's type (engine, transmission, etc.) or its functionality (fan speed control, tire pressure sensing, etc.). Prior work has been able to extract these particular features [54]. Formally, let E_i denote ECU i in a given vehicle that contains n total ECUs that are CAN-enabled. For each E_i in a vehicle's set of ECUs, $E_{1:n}$, the ECU is responsible for sending a specific set of m messages labeled with a unique arbitration ID from the set, $I_{E_i,1:m}$. This set of IDs is unique to E_i and no other ECU in the network should send the same ID. Given a CAN traffic dump from a vehicle, *CANvas*'s enumerator should determine the number of ECUs, n, and differentiate between them to determine the set of ECUs $E_{1:n}$ for that particular vehicle.

Message source identification: Any changes to the set of transmitted messages for each ECU can pinpoint an unexpected reconfiguration potentially done by a fleet owner or vehicle tuner. Likewise, the adversary can tailor the following kill-chain stages to a target ECU based on the messages it sends. Thus, a critical goal for our tool is to map each message ID to its source ECU. Formally, given a CAN traffic dump from which we extract the set of uniquely-ID'd messages where l is the number of total unique message/arbitration IDs and $I_{1:l}$ is the set of unique IDs, we should be able to determine which ECU E_i sent each unique message. This step is very closely related to ECU enumeration; once we know which ECU E_i that an arbitrary ID I_j originates from, we can produce a mapping of the ID to its source ECU, $I_j \in E_i$. Using this mapping, we can group the IDs with a common source ECU and complete our enumeration.

Message destination identification: The adversary needs to ensure that any messages the compromised infotainment ECU will send can be correctly received by the target safety-critical ECU in the following kill-chain stages. This component simply ensures that the adversary faces no surprises when selecting a new target make and model vehicle. We assume that at least one ECU in the network will correctly receive each message in the network. Formally, given the set of l unique IDs, $I_{1:l}$, from a traffic dump, we should be able to determine the set of ECUs, $E_{1:k}$, that correctly receive a message labeled with an arbitrary I_j. The expected output of this component should be a mapping of an ID to its destination ECUs, $I_{j,E_{1:k}}$.

3.3.1 Challenges in an automotive context

To achieve these mapping goals, we encounter two major challenges to determining the source and destination ECUs for CAN messages: (a) CAN lacks identifying source information and (b) CAN implements a broadcast protocol, which naturally implies that all nodes receive all messages.

Lack of source information: If a message sent from ECU E_i has no identifying information, then it is non-trivial to determine that E_i sent the message. Since CAN messages are considered to be "contents-addressed" [68, 69, 70, 39], the value of the message ID is only related to the message's data and priority. In practice, the source ECU has no weight in determining the chosen arbitration ID for a particular message.

Broadcast protocol: We define destination as an ECU that correctly receives a message at the CAN controller level. Unfortunately, determining which ECUs correctly receive a message is non-trivial as an ECU connected to the CAN bus cannot detect which of its messages are received by certain ECUs. The ACK bit itself only indicates that some ECU has received the message, not which particular ECU(s) have received it. As multiple ECUs will set the ACK bit when a message is received, we cannot simply use this ACK bit to determine the set of ECUs $E_{1:k}$ that receive an arbitrary I_j.

3.3.2 System overview

We split *CANvas* into two main components: (1) a source mapper and (2) a destination mapper. As mentioned above, we can satisfy our ECU enumeration requirement by simply using the output of source mapping. For (1), we passively collect several minutes of CAN traffic. After a passive data collection, the source mapper uses the data to produce a mapping of each unique CAN ID to its source ECU and subsequently, by grouping IDs with a shared source, a list of all active source ECUs on the bus. For (2), we interact with the stock vehicle's network directly and perform an active analysis to determine message destination. *CANvas* systematically isolates each ECU, which will most likely cause the vehicle [6] to enter a temporary error state that the adversary can reset.

We assume that the adversary has access to the OBD-II port of the stock vehicle and can connect the *CANvas* mapper directly to the CAN bus with the ability to read and write to the bus. We also assume that the vehicle even has a CAN bus and that the standard CAN protocol is implemented, which most vehicles will reflect [15]. Our source mapper should also only involve software operations so that the adversary can do another mapping once in the victim vehicle's network. For destination mapping on the stock vehicle, the adversary should also be able to transition the vehicle's ignition switch between the LOCK, ACC and ON positions as they will have to reset the vehicle after each iteration to exit the error state.

As seen in Figure 12, the *CANvas* workflow obtains source mapping results by step 4. Then, it will enumerate the ECUs in step 5. *CANvas* then performs destination mapping and generates the full map at step 10. This workflow can be reduced into four major steps:

1. *Data collection:* The CAN pins of the OBD-II port provide access to the frame-level signals

[6] As mentioned earlier, this step would not be performed on the victim vehicle.

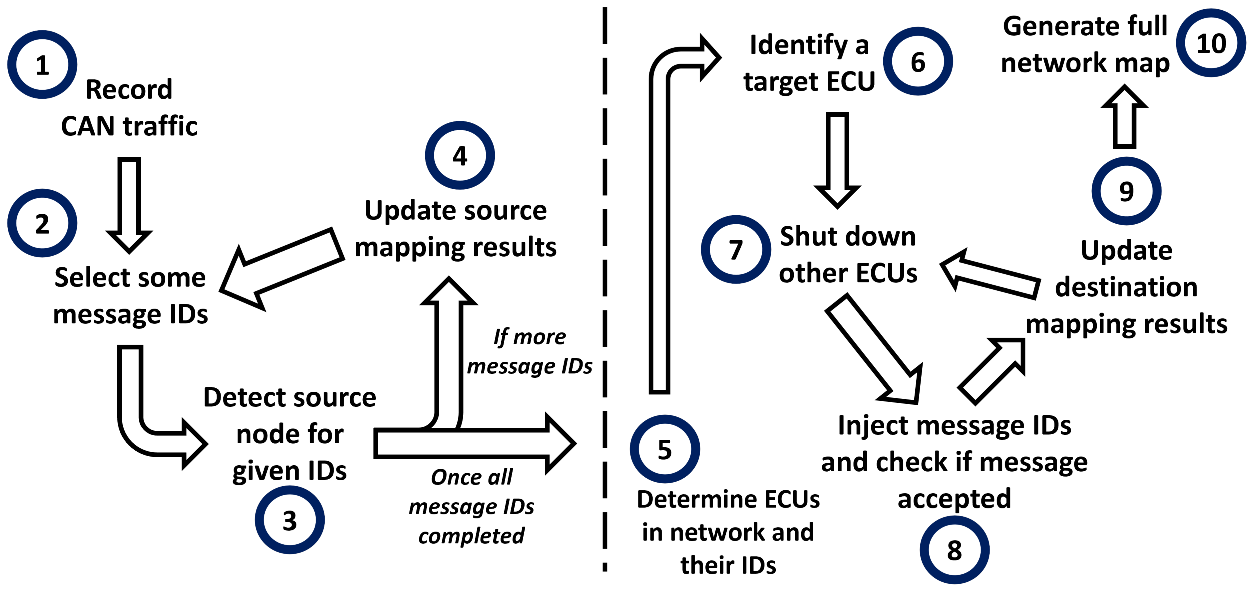

1
Record CAN traffic
2
Select some message IDs
3
Detect source node for given IDs
4
Update source mapping results
If more message IDs
Once all message IDs completed
5
Determine ECUs in network and their IDs
6
Identify a target ECU
7
Shut down other ECUs
8
Inject message IDs and check if message accepted
9
Update destination mapping results
10
Generate full network map

Figure 12: ***CANvas*** **workflow**

and the message-level data. *CANvas* will read this traffic for several minutes and timestamp each received message. From this traffic, we will obtain the set of unique message IDs observed in the network and a set of timestamped data for each ID. Likewise, when on the victim vehicle, the adversary will use the compromised ECU to read timestamped CAN messages as input to source mapping.

2. *Source mapping:* With the list of all unique message IDs, the source mapper will extract the timestamped CAN traffic for each ID and determine which IDs share the same source. To do this, we select two message IDs and run their CAN traffic through our comparison algorithm, which will determine if the two IDs originate from the same ECU.

3. *ECU enumeration:* Using the set of matching ID pairs from source mapping, the enumerator will simply group pairs that originate from the same ECU. The output of this step will be a list of ECUs and associated source IDs.

4. *Destination mapping:* Using the ECU enumeration output, the destination mapper will identify the ECUs that correctly receive a given message ID. *CANvas* will isolate a target ECU by performing a shutdown on all other ECUs. Once an ECU is isolated, we inject all unique

observed message IDs and determine which ECUs receive the message. Again, this step is only performed on the stock vehicle and not on the victim vehicle.

3.4 ID source mapping

In this section, we describe an approach to map each CAN message to its source.

Periodicity exposes identity of source ECU: Due to the absence of source information in a CAN message, we must rely on some uniquely identifying characteristic that can be tied to a particular ECU. Although ECUs communicate on a broadcast bus with no source information in their messages, we find that the periodic nature of their transmissions can permit an adversary to uniquely identify a message's source. Following observations from prior work [15, 43] and CAN documentation [39, 71], we consider *clock skew* as a candidate fingerprinting mechanism. In particular, time instants for in-vehicle ECUs rely on a quartz crystal clock [39], and we can use the relationship between these clocks to identify a transmitting ECU. We first define the following terms considering two clocks, $\mathbb{C}_1$ and $\mathbb{C}_2$:

- **Clock frequency**: The number of cycles per true second, e.g., if $\mathbb{C}_1$ operates at 16kHz, then $\mathbb{C}_1$ cycles 16,000 times every one true second.
- **Relative clock offset**: The difference in time reported by $\mathbb{C}_1$ and $\mathbb{C}_2$, e.g., if $\mathbb{C}_1$ reports time t_1 of 4.1ms and $\mathbb{C}_2$ reports t_2 of 4.2ms, their offset $O_{\mathbb{C}_1,\mathbb{C}_2}$ is 0.1 ms. Where only one clock is denoted for relative offset, the other clock is the clock of the receiving node.
- **Relative clock skew**: The difference in clock frequencies of two clocks, or the first derivative of offset w.r.t. true time, e.g., if $\mathbb{C}_1$ operates at 16kHz and $\mathbb{C}_1$ operates at 16.1kHz, their skew $S_{\mathbb{C}_1,\mathbb{C}_2}$ is 100Hz. Where only one clock is denoted for relative skew, the other clock is the clock of the receiving node.

Two clocks with a relative clock offset of 0 are considered to be *synchronized*, and two clocks with a nonzero relative clock skew are said to "skew apart," or have an increasing relative offset over time [71]. Since the CAN protocol does not implement a global clock, it is considered to be unsynchronized as each ECU relies on its own local clock. The clock offset and skew of an ECU relative to any other ECU is distinct, thus providing us with a uniquely identifying characteristic for source mapping.

High-level idea: To map each unique ID to its transmitting ECU, we break the module into two steps: (1) computing either the skew $skewf(I_i)$ or offset $offsetf(I_i)$ of each ID I_i and (2) then clustering IDs with the same skew or offset where each cluster denotes a distinct source ECU, E_{src}. This module outputs a mapping of source ECUs to their set of source IDs. The main input to this module is a passively-logged CAN traffic dump, which contains entries in the form of $(I_i, t_{I_i,n})$ where I_i is the ID of the message and $t_{I_i,n}$ is the timestamp of the n^{th} occurrence of I_i.

3.4.1 Limitations of prior work

Cho et al. use clock skew as a means of building an intrusion detection mechanism to identify an attack by a malicious ECU [15]. Specifically, this work uses timestamps of periodically-received message IDs and posit that IDs with the same skew originate from the same ECU.

To compute the clock skew of an ID I_i over time, Cho et al. perform the following steps [15]: (1) compute I_i's expected period, μ_{T_i}, (2) compute the offset, O_i, by subtracting the expected timestamp (using μ_{T_i} from the actual timestamp), (3) take the average of O_i over a batch of N messages, (4) add $O_{i_{avg}}$ to an accumulated offset, O_{acc}, and (5) then compute the skew, S_{I_i}, by taking the slope of O_{acc} versus time. This work uses the Recursive Least Squares algorithm to minimize the errors. After every batch of N messages, O_{acc} increases by O_i, where k is the k^{th} batch. From this plot, since O_i should be constant, their formula for skew w.r.t. batch size sets S_{I_i} to:

$$skewf_i^{Cho}(N) = \frac{kO_i}{kN} = \frac{O_i}{N} \tag{1}$$

As an extension to this work, Sagong et al. note that the skew of Equation 1 varies significantly based on N and use an updated formula for S_{I_i} w.r.t. batch size [43]:

$$skewf_i^{Sagong}(N) = N \cdot \frac{kO_i}{kN} = O_i \tag{2}$$

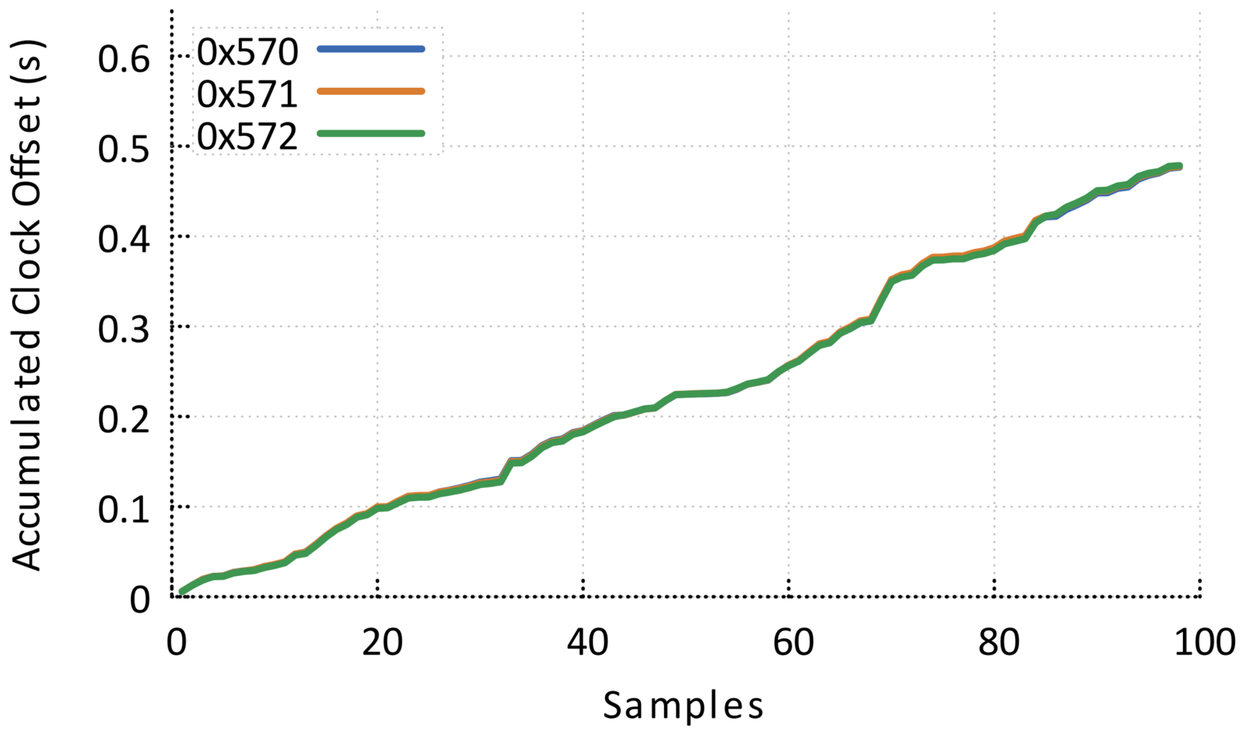


Figure 13: Prior work mapping an ECU with same periods

Using data from a real vehicle, we now highlight a key limitation of Equations 1 and 2. Consider Figure 13, where E_A is the source of IDs 0x570, 0x571 and 0x572, which share the same period and then, consider Figure 14, where E_B is the source of IDs 0x262, 0x4C8 and 0x521, which each have different periods. In Figures 13 and 14, we use the equation from prior work [15], $skewf_i^{Cho}$, with N = 20 to plot the skew of all six IDs; $skewf_i^{Sagong}$ produces similar results. We can correctly conclude from Figure 13 that the IDs of E_A originate from a single ECU. However, from Figure 14, we will incorrectly conclude that IDs 0x262, 0x4C8 and 0x521 originate from three *separate* ECUs. Our analysis and experiments shed light on why these approaches fail; the skew value they compute is *period-dependent.*

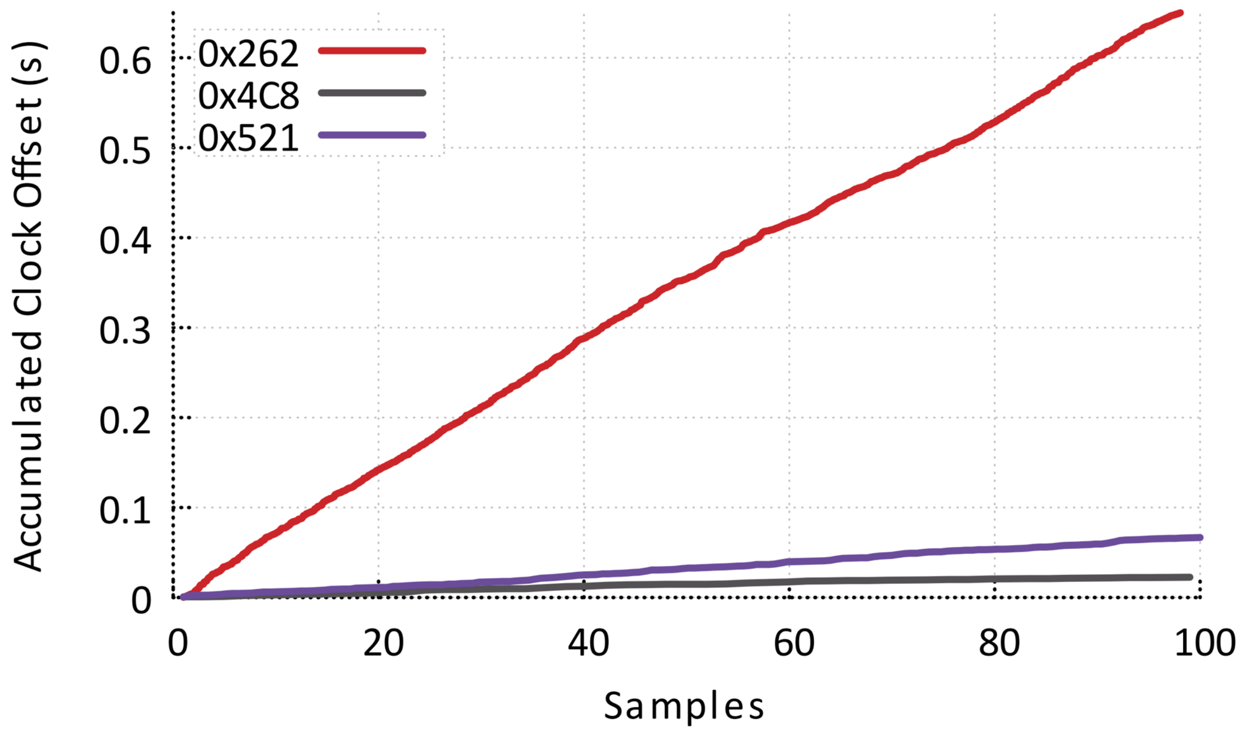


Figure 14: Prior work mapping an ECU with different periods

As such, we update Equations 1 and 2 w.r.t. period T and batch size N:

$$skewf_i^{Cho}(N,T) = \frac{kO_i}{kTN} = \frac{O_i}{TN} \tag{3}$$

$$skewf_i^{Sagong}(N,T) = N \cdot \frac{kO_i}{kTN} = \frac{O_i}{T} \tag{4}$$

To potentially fix this issue, we can attempt a strawman that is not dependent on period or batch size.

$$skewf_i^{Straw}(N,T) = TN \cdot \frac{kO_i}{kTN} = O_i \tag{5}$$

Ideally, accounting for both batch-size and message-period (essentially batch-period, NT) using Equation 5 should give us a unique value that is common only among IDs from the same ECU.

We apply Equation 5 for all I_i of a vehicle, and we attempt to establish distinct groupings of the computed skew for each ID, S_{I_i}, which would identify which I_i share the same E_{src}.

Unfortunately, this is a difficult task as I_i from the same E_{src} still do not have similar skews. This issue is further demonstrated as S_{I_i} varies across different data dumps or even segments of a given dump. Upon further inspection, we find that the measured S_{I_i} is affected by the deviation in an ID's period. This deviation in the period, σ_{p_i}, is attributed to sources of "noise", i.e., the period of a given message varies due to scheduling, queuing and arbitration delay. We also find that some I_i produce S_{I_i} with more deviation than others and produce widely-varying skew values, thus making our straw-man solution an unlikely candidate for source mapping. It is clear that we now need a method of extracting the clock skew invariant that is: (a) independent of the period of I_i and (b) robust to noise in the period.

Issue with straw-man: In Equation 5, it is clear that, relative to the receiver, this "skew" function computes offset rather than true skew. Following our definitions for source mapping, a plot of relative offset over time should either be linearly increasing or decreasing if there is a nonzero skew between two clocks. In other words, if the relative skew between an E_{src} and the receiver is non-zero, then we should observe a gradual change in the offset. However, previous work fails to capture this change in offset over time [15, 43].

3.4.2 Relative offset as a unique identifier

As mentioned above, clock offset and skew of an ECU relative to another ECU is distinct. We must note that the clock offset measured from one ID, I_1, of an E_{src} may not be the same as the offset of another ID, I_2, from E_{src}. If the initial transmission time of I_1 differs from that of I_2, the O_{I_1} could not equal O_{I_2}. Rather, the invariant here is the *change in relative offset*, ΔO_{I_i}; as the skew of E_{src} relative to the receiver is a constant nonzero value, the ΔO_{I_i} will be a constant nonzero as well (the derivative of offset is skew).

By measuring this change in offset, we can uniquely identify an E_{src}, but we must ensure our method of extracting this change in offset is (a) robust to a noisy period and (b) period-independent. To address the issue of noise in the period of I_i, p_{I_i}, we compute the relative offset between a pair of two different IDs denoted by O_{I_1,I_2}. By performing this computation pair-wise, we expect O_{I_1,I_2} to have a deviation of approximately 0 if $I_1, I_2 \in E_{src}$ as the sources of noise for I_1, I_2 should mostly be shared. In reality, this deviation is very close but not exactly equal to 0; we define a practical threshold for this deviation in our evaluation.

With a pairwise approach to computing O_{I_1,I_2} and the requirement for a period-independent approach, we face a new challenge: determining at what point in time to observe this relative offset regardless of the period of I_1 or I_2. Thus, we can compute offset at the least common multiple of I_1 and I_2, which we call the *hyper-period*.

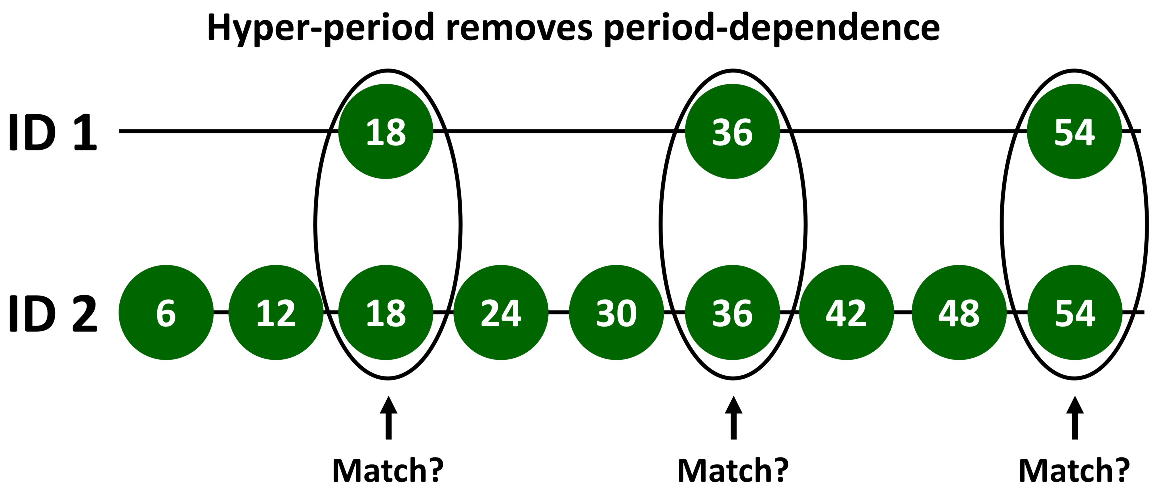


Figure 15: Measuring clock offset at the hyper-period

Measuring offset at the hyper-period: To guide our algorithm design for computing $\Delta O_{I_1,I_2}$ over time, we first model two periodically-transmitted IDs observed on the CAN bus. Consider two IDs, I_1 and I_2, from the same E_{src} which transmit at a period of p_1 and p_2, respectively. For example, let p_1 be 18ms and p_2 be 6ms. For now, we assume that the relative offset between I_1 and I_2 is 0. This offset should not change over time as they originate from the same E_{src}. To accurately compute the relative offset of these two IDs, O_{I_1,I_2}, we must select a time instant when the expected offset should also be 0: the hyper-period of I_1 and I_2, or the least common multiple of p_1 and p_2. As seen in Figure 15, this time instant occurs at 18ms, or the $lcm(18, 6)$. Therefore, by computing the difference between the times reported from I_1 and I_2 every 18ms, or the hyper-period of I_1 and I_2, we can track the value of relative offset over time. If this relative offset is a nonzero constant, then the two IDs originate from the same ECU.

With an input of several minutes of timestamped CAN data, we can track relative offset over the timeline of two message IDs. Note that each timestamp has a noise component that stems from scheduling, queuing and arbitration delay. To compare whether two message IDs originate from the same ECU, we first assume that they are sent by separate ECUs. The two message IDs, I_1 and I_2, have periods, p_1 and p_2, and they have relative offsets, O_{I_1} and O_{I_2}. We draw the following relationships between these variables:

- $p_2 = lp_1$, where l is the ratio of the periods.
- $O_{I_2} = jO_{I_1}$, where if j=1, then both IDs sent by same ECU; otherwise, they were sent by different ECUs.
- $n = ml$, where $LCM(n, m) = l$ as depicted in Figure 15.

By computing the difference between every n occurrences of I_1 and every m occurrences of I_2, which occurs at the *hyper-period* of I_1 and I_2, we produce the following equation:

$$O_{I_1,I_2} = (mp_2 + O_{I_2} + i_2) - (np_1 + O_{I_1} + i_1) \tag{6}$$

We find that when we average the result of Equation 6 across the entire data log, the expected value is 0 if I_1 and I_2 originate from the same ECU. In reality, this value is close to 0 due to the deviation of a message's period. From experimental data, we define a threshold of 1ms for the change in relative offset, where a value under the threshold will classify the two IDs with the same source ECU. Using this approach to revisit the setup described in Figure 14, we correctly conclude that IDs 0x262, 0x4C8 and 0x521 originate from the same ECU.

Practical challenges: While the above approach is correct, there are a number of other practical challenges we need to address to ensure accurate mapping:

1. *Large hyper-period*: Consider a hyper-period that is "large," or on the scale of several minutes, e.g., the hyper-period of $p_1 = 980$ms and $p_2 = 5008$ms is over 20 minutes. Since we only extract one relative offset value per hyper-period, we would need hours of CAN traffic to produce a valid result. To ensure that our mapper is fast, this length of traffic log becomes unreasonable; we want to produce a full network map in under an hour. Fortunately, with a pairwise approach, we can choose to *not* attempt a comparison when the hyper-period is large; for example, if we assume that the E_{src} of I_1 also transmits another ID, I_3, where the hyper-period of I_1 and I_3 is small, we can still determine that $I_1, I_3 \in E_{src}$.

2. *Large period deviation, σ_{p_i}*: In early experiments, we discovered messages that had a large measured σ_{p_i} (we define large as $\sigma_{p_i} \geq 0.1p_i$) and, at first, assumed that these messages were either aperiodic or sporadic (aperiodic with a hard deadline). However, upon closer inspection, we noticed that these messages appeared to be periodic in nature. We observed three different patterns that altered the measured σ_{p_i}: (1) the period simply had a large σ_{p_i}, (2) periodic messages would occasionally stop transmitting for some time, and (3) periodic messages were missing their deadlines. With a large enough σ_{p_i}, the deviation would conceal an inconstant ΔO_{I_i} and make it difficult to detect a mismatch. We experimentally find that a σ_{p_i} greater than 8% of p_i results in incorrect outputs. Therefore, *CANvas* will choose to test I_i on the following cases when its σ_{p_i} is under a defined threshold, which we set to $\sigma_{p_i} \leq 0.08p_i$ from our experiments.

3. *Periodic messages that occasionally stop:* We find that some I_i are periodic and will stop transmitting for some time, causing a measured σ_{p_i} to be large. To combat this issue, we only perform pairwise offset tracking when the given message was actively transmitting. In the event we compare two I_i that both occasionally stop and there is no overlap of active transmissions, we then rely on our pairwise approach to match the I_i to another ID from the same E_{src}.

4. *Messages that miss deadlines*: For some I_i with a large σ_{p_i}, we observe two different inter-arrival times: p_i and $2p_i$. When a task on one of the ECUs misses its deadline and cannot produce a message on time, it will skip that cycle and transmit during the next cycle [71]. Thus, when a deadline is missed, we will observe an inter-arrival time of $2p_i$. In this situation, there are two options: (1) perform relative offset tracking on portions of the log when deadlines are not missed or (2) interpolate the missed inter-arrival times. If a message frequently misses its deadline, the first option is not viable. To interpolate a missed arrival time, we insert an entry in the traffic log with a timestamp equal to the average of the preceding and the following timestamp.

Factors for mapping time: For source mapping, we experimentally find that 30 minutes of data provides enough samples for larger hyper-periods to map accurately. While this stage has static run-time, the variation in time requirements will be dependent on the number of observed messages IDs. The more message IDs that exist in the network, the longer the mapping time takes; vehicles with more message IDs take longer to complete mapping due to an increase in message-pairs. However, to further reduce mapping time, mapping messages with small periods requires much less traffic data. To save additional time if necessary, it is recommended to reduce the traffic log length for high-frequency messages. Also, if there are few large periodic messages or if those messages are not relevant for whatever reason, the length of the initial traffic log can be reduced as necessary instead of the recommended 30 minutes.

3.5 ID destination mapping

While ID source mapping is the primary contribution of this kill-chain stage, we also want to provide destination information about each ID in a CAN bus. As our approach to destination mapping requires physical access, we can perform this mapping on the stock vehicle to ensure that a vehicle's ECUs does not block any messages at the hardware-level.

Intuition: The destination(s) of a particular CAN message are those ECUs who correctly receive a given message. Despite the broadcast nature of CAN, if an ECU does not correctly receive a message, it will not set the ACK bit; however, if other ECUs receive this message, they will set the dominant ACK bit. Unfortunately, an ACK observed by the transmitting ECU only means that *some* active ECU correctly received the message. Therefore, with multiple active ECUs in the network, we cannot identify which ECUs were the destination for a given message. Consider the scenario where there is only one active destination ECU, E_{dst}, in the network other than a transmitting source ECU, E_{src}. For each message sent by E_{src}, a set ACK bit (performed only by E_{dst}) would indicate that only one ECU received the message: E_{dst}. Thus, in this scenario, E_{src} could simply inject all possible I_i and detect which messages have a set ACK bit. The major challenge here is identifying a method of isolating an E_{dst} and "removing" all other ECUs from the network. We define the bare minimum of "removal" as preventing an ECU from participating in the acknowledgement process. Thus, our idea for performing this removal is to transition an ECU into an error-state that prevents it from setting the ACK bit for any message. We can do this by exploiting the error-handling mechanism in CAN as detailed in Chapter 2.

3.5.1 Limitations of prior work

Imposing bus-off state: The challenge in transitioning an ECU to bus-off is to determine what kind of error to produce and how to produce it. We look to previous work [30] that aims to shut down an ECU for the purpose of an attack. The authors aim to shut down an ECU by causing an error in the target ECU. By exploiting the error-handling protocol in CAN, where bus-off effectively removes an ECU from the network, they choose to increment the error counter of a target by causing a bit error. This error occurs when a transmitting ECU reads back each bit it writes; when the actual bit is different, the ECU invokes an error. Since only one ECU is expected to win the bus arbitration, the authors point out that two winners would potentially cause a bit error. For example, suppose that the victim ECU transmits a message with ID 0x262. If the attacker ECU also transmits ID 0x262 at the exact same time as the victim, both ECUs will win arbitration. However, to ensure that the victim has a bit error, the attacker's message will set its DLC, or data length count, to 0 (most practical messages contain at least some data). After a sufficient number of these attack messages, the victim ECU will transition into the bus-off state.

The main challenge here is synchronizing the attack message with the victim message so they both enter arbitration simultaneously. Their insight is to inject a message of higher priority around the time when the victim should transmit. The higher priority message will block the victim until the bus is idle, where it will then transmit. The attacker will load its attack message immediately after the higher priority message is transmitted, thus allowing both the victim and attack message to arbitrate simultaneously. Since there is noise in the true transmission time of the victim's first attempt at transmitting, there is a chance that the attacker will need to make multiple attempts to cause an error. The number of injection attempts needed to cause a single bit error, κ, is defined as the following where $\mathbb{I}$ is a confidence attack parameter (high parameter value means higher confidence in attack), σ_{p_v} is the jitter deviation of the victim's period, and S_{bus} is the speed of the bus in Kbps:

$$\kappa = \left\lceil \frac{2\sqrt{2}\mathbb{I}\sigma_{p_v}S_{bus}}{124} \right\rceil \tag{7}$$

The authors state that only one of these injections is needed to cause a bit error if setting $\mathbb{I} = 3$ and at most 2 if setting $\mathbb{I} = 4$, given that the period deviation is 0.025ms.

Straw-man limitations: Suppose we used the above approach to cause a bus-off in a real vehicle. Unfortunately, in sample traffic dumps from two real vehicles, the smallest deviation that we observed was approximately 0.15ms. Using the equation given by Cho et al. [30], the number of preceded message injections per error is 8 when the period deviates by at least 0.205ms; if 8 injections are required, any successful bit error would be undone by successful message transmissions. We look at available traffic logs used in the works by Miller et al. [51]. For this traffic log, the majority of the messages have a period deviation over 0.205ms. In other words, assuming the best-case scenario of 0.15ms, we would need to inject at least 6 higher-priority messages, or preceded messages, for a bus speed of 500Kbps. Considering that each successful transmission by the victim ECU decrements the error count by 1, we would effectively only increase the error count by 2 with each successful attack (instead of the expected 8). Since the majority of messages have a period deviation greater than 0.205ms, it is highly unlikely to use this method for isolating an ECU. Thus, we need a method of transitioning an ECU into the bus-off state that is reliable and robust even when the period deviates by more than 0.025ms.

3.5.2 Forced ECU isolation

High-level idea: To map each unique ID to its set of destination ECUs, we break the module into two steps. We repeat these two steps for all n ECUs in the network. The first step is to isolate the target ECU and shut off all others by transitioning the non-target ECUs to the bus-off state. As there are n ECUs in the network, we will need to "bus-off" $n-1$ ECUs for each ECU (i.e., we will need to perform the bus-off at least $n(n-1)$ times). Once we isolate an ECU, we then inject the set of all I_i and observe which messages have a set ACK bit, thus identifying the set of I_i where the target ECU is an E_{dst}.

Inducing a direct bit-error: Isolating an ECU via the bus-off method requires a quick and effective approach. Since we are not limited to operating through the interface of a CAN controller, we can directly view the CAN frames in real-time via digital I/O pins. However, since we are using a microcontroller that operates at the same voltage of the CAN controller, we do not operate at the true CAN voltage. Instead, we tap directly between the interface of the Arduino's CAN controller and the CAN transceiver, where we can safely access the bus data. At this junction, we observe that the data on the line is within the Arduino's voltage and contains the full data frame, including SOF, ACK and EOF bits. With this access to the full data frame rather than just the components of the CAN message, we can directly induce an error on the bus and thus achieve the bus-off attack. By reading the ID of the message in real-time, we can choose to attack any ID by simply driving a dominant bit to the CAN transceiver.

Note that the bus-off method requires attacking a message ID every time it occurs until the ECU enters the bus-off state. However, in the event that a message has a very long period, the time to perform the bus-off will take significant time. As such, we can employ the result of *CANvas*' source mapping component by identifying the ID with the smallest period per ECU and attacking just that ID. In practice, we have found that every real ECU we have encountered has at least one ID that operates under 100ms. Thus, this approach makes the destination mapping component of *CANvas* fast.

Determining message receive filter: Now that we can isolate a single ECU in the network, we can simply inject all messages in the observed ID space and determine which messages are correctly received by the ECU. However, to view the ACK bit at the network level, which is not visible to the user, the obvious option is to use a logic analyzer. For simplicity, we seek an alternative that uses the same Arduino. We observed that, if a message is sent to a single ECU and it does not correctly receive the message, the transmitter will re-attempt to send the message until it is received correctly. As such, if we transmit a message and see a continuous stream of the same ID from our mapping device, then we may conclude that the message ID is not received by the isolated ECU.

Practical challenges: Since our approach to destination mapping involves shutting off multiple ECUs at a time, we encounter a couple of challenges in a real vehicle setting: (1) ECUs that auto-recover and (2) ECUs that are persistently active. We now define these scenarios and provide a detailed approach to addressing these practical challenges:

1. *ECUs that auto-recover*: In our earlier experiments, we performed a simple experiment to verify the potential of an isolation method. We attempted to transition all ECUs in the network to the bus-off state by shorting the CAN bus pins, which would effectively cause a transmit error for all ECUs and force them into bus-off. However, after removing the short, we saw that some CAN messages were still transmitted onto the network, clearly indicating that some ECUs left the bus-off state. We find that these ECUs would wait a predefined amount of time before re-transmitting again as these ECUs were critical to the vehicle's

powertrain (engine, hybrid, etc.) [39]. In this situation, we would transmit a portion of the injected messages onto the bus and then re-isolate our target ECU when a non-target starts to transmit again. This approach is only reasonable for recovery times on the scale of seconds.

2. *ECUs that are persistently active*: Out of the set of ECUs that did auto-recover, we also noticed that one ECU seemed to be persistently active. In other words, there appeared to be no delay between a transition into the bus-off state and the next transmission from the ECU. Upon closer inspection, we found that this ECU would auto-recover only after 128 occurrences of 11 recessive bits [40]. In this situation, we must "hold" the bus open by constantly transmitting false messages from our device to trick the recovering ECU into thinking that the bus is still active.

Factors for mapping time: For destination mapping, the run-time is dependent on the number of ECUs and increases with more ECUs. We acknowledge the potential of long run-times for vehicles with tens of ECUs if all were CAN-enabled. To combat this, we suggest performing the bus-off on the ID with the smallest period per ECU to reduce the time attributed to achieving ECU isolation. Also, for our two vehicles, all observed IDs were active when the vehicle was simply in ACC rather than ON so there may be no need to crank the engine per ECU.

3.6 Evaluation

We now present our evaluation for the *CANvas* network mapper and detail the following results:

1. identifies an unexpected ECU in a '09 Toyota Prius,
2. identifies the absence message-receive filters in a '17 Ford Focus,
3. produces a sound source mapping of two real vehicles and accurately identifies the source of approximately 95% of all I_i in the network and a complete destination mapping with an isolation technique that is 100% reliable,

4. successfully demonstrate our forced ECU isolation on three extracted ECUs,

5. and produces source mapping of three additional vehicles.

Setup and methodology: Our experimental setup includes five real vehicles and several synthetic networks to demonstrate the above benefits. Below is a brief description of the *CANvas* hardware implementation, five real vehicles and our synthetic network of real ECUs:

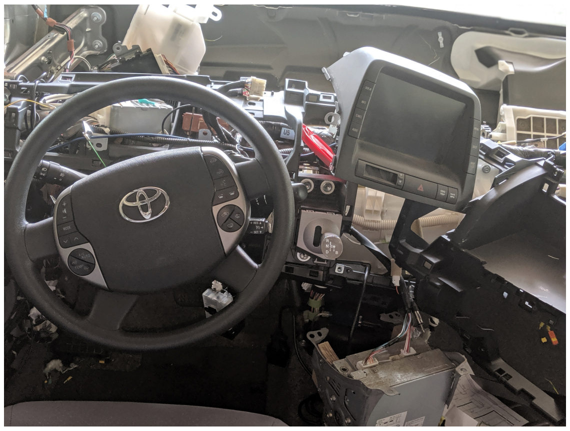


Figure 16: A thoroughly disassembled 2009 Toyota Prius

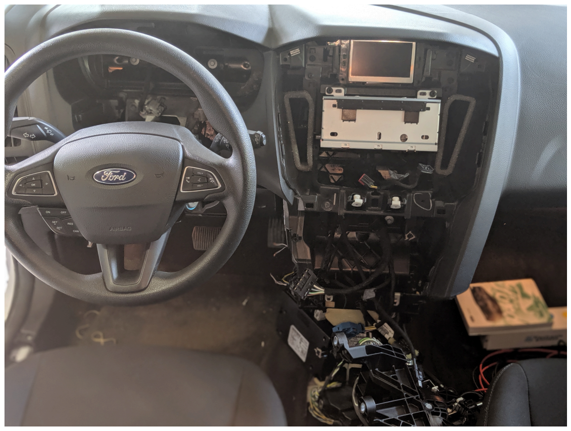


Figure 17: A thoroughly disassembled 2017 Ford Focus

- *Mapping device:* To interface with a CAN bus, our mapping device consists of three components: an Arduino Due microcontroller with an 84 MHz clock and an on-board CAN controller, a TI VP232 CAN transceiver, and a 120Ω resistor. To gain direct write access to the bus for destination mapping, we connect a digital I/O pin to the driver input pin of the transceiver.

- *'09 Toyota Prius and '17 Ford Focus:* The Prius contains eight original ECUs that transmit on a single CAN bus at 500 kbps. The Focus contains eleven original ECUs that transmit on three CAN buses at varying speeds; as our model of the Focus is the standard edition, only the high-speed 500 kbps bus has more than one active ECU. We obtain ground truth for our experiments by physically taking apart the car and gaining direct access to the ECUs by splicing directly into the CAN wires as seen in Figures 16 and 17. We use a paid subscription to both Toyota and Ford's mechanics' manuals for guidance on disassembly of

ECU #	*Source message IDs*	*Actual ECU*
A	020, 030, 0B1, 0B3, 0B4, 230, 4C3, 591	Skid control ECU
B	022, 023	Yaw rate sensor
C	025, 4C6	Steering sensor
D	038, 03A, 03E, 120, 244, 348, 527, 528, 529, 540, 5B2, 5C8, 5EC, 602	Hybrid control ECU
E	039, 3C8, 3CF, 526, 52C, 5CC, 5D4, 5F8	Engine control module
F	262, 4C8, 521	Power steering ECU
G	3C9, 3CB, 3CD	Battery ECU
H	553, 554, 57F, 5B6	Gateway ECU
I	570, 571, 572	*Unknown ECU*

Table 2: 2009 Toyota Prius source mapping output

vehicle components [72, 73]. Due to the non-destructive design of *CANvas*, our interaction does not impose any permanent errors to the vehicle.

- *'08 Ford Escape, '10 Toyota Prius, and '15 Ford Fiesta:* We obtain CAN traffic from three additional vehicles for testing only our source mapper, as we did not have permission to inject data. Since the vehicles are from the same make, we use data from the '09 Prius and '17 Focus to partially confirm our source mapping output without disassembling these vehicles.
- *Synthetic networks:* To further validate the capability of our mapper, we perform additional experiments on three real engine ECUs extracted from a '12 Ford Focus, '13 Ford Escape and '14 Ford Escape.

3.6.1 Discovering an unexpected ECU

We now describe a real scenario where, in the process of designing *CANvas*, we discovered an unexpected ECU in our Prius. Using the results of our source mapping on the '09 Prius as seen in Table 2, we noticed that there was a total of nine ECUs when only eight were expected. Even after manually disconnecting all eight known ECUs, we still observed CAN traffic, specifically IDs $I_{570-572}$, coming from a single ECU. By looking at the history of the vehicle and systematically disconnecting various systems, we discovered that this ECU was installed as part of a modification from several years ago. The Prius had an additional battery installed to grant it all-electric capabilities and, with the use of the network mapper, we now know that a new CAN-enabled device was added. We confirm that these IDs are new by comparing our IDs to a same-generation Prius [74]. Thus, if we took a network map of a stock 2009 Toyota Prius, we could easily compare our results on a victim vehicle and find that the victim network was different.

3.6.2 Mapping our test vehicles

Using the results of *CANvas*' destination mapping, we can identify several instances where an ECU is expected to only receive messages from a subset of other ECUs but still receives all other messages. We have found that all ECUs in the Focus and Prius do not employ any filter on the receipt of incoming messages. In Ford's Motorcraft TechInfo Service, we can see simple diagrams of how the ECUs communicate as part of the vehicle's systems [73]. For example, the Focus' braking system involves communication between the instrument panel cluster, the transmission ECU, the body control ECU, and the engine ECU.

Source mapping results: Using a threshold of 1ms and 30 minutes of traffic collection, we get a false positive rate of 0% for both vehicles, permitting us to get a sound source mapping output. Out of a total of 59 unique message IDs, our pairwise timing comparison resulted in 102 matching pairs for the Prius. By performing a simple grouping of these pairs, we get the output as seen in Table 2. While the majority of the IDs observed on the Prius have a strong periodic characteristic, we discuss some special cases we encountered. Most of the messages were under five seconds except for I_{57F} with a period of 5 seconds and I_{602} with a period of 60 seconds. The majority of our messages matched with multiple IDs from the same ECUs but due to the large period of I_{57F} and I_{602}, they only had a single match. However, due to our pairwise approach, we can still map these two IDs using a shared matching pair. We also encounter a few examples of messages that miss their deadline and wait until the next cycle to re-transmit. For the Focus, we observe messages that miss their deadlines and either transmit two messages on the next cycle or drop the missed message and wait for the next cycle. In these cases, we simply remove the inter-arrival times that exceed two standard deviations from the average period and interpolate for the removed timestamps.

Destination mapping results: With a CAN bus running at 500 kbps, we discover that all of the ECUs in the Prius do not implement any filtering between the network and the CAN controller. When each ECU is isolated, we see that all IDs are properly acknowledged by the receiving ECU. We do observe two ECUs that recover quickly from the bus-off method, specifically the engine control module and the skid control ECU. With the other ECUs in the vehicle, it was sufficient to perform our bus-off once and the ECU would stop transmitting. For these two ECUs, we selected the smallest period ID and held the bus open by injecting false messages to keep the two ECUs from auto-recovering. Additionally, we discovered that the Focus also do not implement any sort of filtering for the IDs we observe on the CAN. From these findings, we can conclude that attacking via the reception of a message for these vehicles could prove trivial due to the lack of filtering between the network and the controller. In general, the maximum number of manual transitions of the ignition switch is equal to the number of detected CAN-enabled ECUs in the vehicle. For the keyless ignition of the 2009 Prius, we transition the ignition 7 times as two ECUs recover on their own (the Prius has 9 total CAN-enabled ECUs). For the keyed ignition of the 2017 Focus, we transition the ignition 7 times as two ECUs recover on their own (the Focus has 9 total CAN-enabled ECUs).

Mapping real extracted Ford ECUs: We also obtained three Ford engine ECUs from a ‘12 Focus, ‘13 Escape and ‘14 Escape. By collecting data from these three ECUs, we found that they shared the many of the same message IDs and conclude that they are based off of the same engine controller configuration. As they all auto-recover, they were prime candidates for testing our forced ECU isolation technique.

Mapping other vehicles: We use *CANvas* on three other vehicles to look for data that seems logical to our findings from the test cars. For the Ford vehicles, we look for similarities with our extracted engine ECUs. For the ‘08 Escape, we found a set of IDs that we believe is the engine ECU and only has a subset of those found on our extracted ECU. For the ‘15 Fiesta, we also found a likely candidate for an engine ECU that has more IDs than our extracted ECUs. Since these vehicles range over three different Ford generations, it seems logical that the newer engine ECUs transmit more IDs. Additionally, we find a few similarities between the ‘09 and ‘10 Prius. We found an ECU on the ‘10 that is likely to be the skid control ECU, which has similar IDs to the ’09 Prius. These findings potentially demonstrate *CANvas*’ source mapping capabilities even without ground truth.

3.7 Countermeasures

The core disconnect that we exploit is that CAN messages are periodic in nature and thus expose a unique timing characteristic.

Masking clock offset: One potential defense here is to intentionally add variance to the periodicity so that it would be challenging for *CANvas* to identify which messages originate from which ECUs. The concept of masking unique clock characteristics is not a new defense in the field of traditional network [75]. However, it is highly unlikely that an automaker would intentionally alter message timing (as a method of masking clock characteristics) due to the challenges that arise from scheduling real-time embedded systems. There are numerous challenges that automakers already face in achieving reliable and robust scheduling for their vehicles, and any modification to the timing of CAN messages would add a great amount of complexity to the already complex challenge of scheduling.

Dynamic message IDs: Another option for a defense is to make it challenging for the adversary to track a message by its ID by changing the ID each time a message is sent from the same ECU. However, since the message ID plays a role in enforcing the priority of a message, it is highly unlikely this defense would be practical. Likewise, any encryption on the CAN bus will likely exist for only the data payload and not the message ID as it still plays a role in priority. In short, as the *CANvas* network mapper does not employ any active measures and just passively sniffs the bus for source mapping, it will be difficult to build countermeasures against it.

3.8 Summary

In this chapter, we presented *CANvas*, a network mapper that could identify all transmitting ECUs on a CAN bus. *CANvas* leverages the periodicity of CAN messages to extract unique timing characteristics that can uniquely identify the origin of a message even though these ECUs exist on a broadcast network. By exploiting this disconnect, *CANvas* can accurately map real in-vehicle networks and enables a remote adversary to obtain critical information about a network prior to launching the next attack stages. We also propose countermeasures that target this disconnect by intentionally removing periodicity or making it difficult for an adversary to track periodic messages.

4 *CANnon*: Remote Disruption of CAN Bus via Peripheral Clock Gating Attacks

In this chapter, we discuss our contributions to the disruption and pivoting stage of our attack kill chain. The contributions of the thesis in this stage are:

- **Kill-chain goal:** In this stage, we aim to develop a disruption attack that can impact the transmissions of other ECUs and affect its state on the CAN bus. With this ability, the remote adversary can forcibly delay the transmissions of other ECUs or even impact the synchronization of other ECUs to the CAN bus.
- **Disconnect to exploit:** We find that a new power-saving feature called peripheral clock gating permits software instructions to "freeze" a CAN controller in the middle of a transmission, impacting a CAN message at the CAN physical layer. By disabling the clock with this software instruction at specific times, the remote adversary can even shut down a specific ECU in addition to impacting its transmissions on the CAN bus.
- **Potential countermeasures:** The key vulnerability insight here is that the peripheral clock gating feature can be exploited during the transmission of a CAN message. Thus, we propose countermeasures to either detect the use of clock gating in the middle of a CAN transmission or prevent clock gating from being exploited (even by removing this feature altogether).

4.1 Goals for this kill-chain stage

As our adversary cannot directly compromise the safety-critical target ECU (due to being limited as a remote adversary), they must aim to utilize the initial compromised ECU to influence the functionality of the target ECU (and its CAN transmissions) without being detected by any deployed network security mechanisms. One advantage our attacker can exploit is the recent design choice in modern high-performance ECUs, where the microcontroller units (MCUs) implement peripheral clock gating for the CAN controller. By exploiting a vulnerability enabled by this design, our attacker can use the compromised ECU to influence how other ECUs interact with the CAN bus. To demonstrate this, we detail a standalone attack that focuses on shutting down a target ECU. However, the methods that enable this shutdown attack can be used to extract information in the next kill-chain stage by controlling when messages appear on the bus and by controlling how other ECUs synchronize their timing to the CAN bus.

Typically, the lack of security on in-vehicle CAN buses used to permit an adversary with access to the CAN bus to arbitrarily insert, modify, and delete messages, allowing an attacker to manipulate the functionality of safety-critical ECUs [8] or limit communication over the bus [46, 47]. However, as a defense against an evolving threat landscape, academic and industry researchers have proposed a variety of techniques, such as message authentication [76, 77], intrusion detection systems (IDSes) [48, 16, 17, 29], and secure CAN hardware solutions [20]. Thus, we aim to investigate a new disruption attack for an adversary to impact CAN bus communications that does not rely on CAN message injection.

4.2 Stage overview and contributions

Despite efforts to increase the security of automotive networks, a recent class of attacks demonstrates significant adversarial potential by utilizing the inherent CAN protocol framework to *shut down* safety-critical ECUs. Such attacks introduced by prior work are particularly dangerous due to their ability to disable critical vehicle functionality by shutting down several ECUs from just a single compromised ECU [46, 30, 47]. Additionally, an adversary could use shutdown attacks to launch advanced attacks (e.g., stealthy masquerade attacks [15, 43]). Current shutdown attacks repeatedly trigger the error-handling mechanism on a victim ECU, causing it to enter the *bus-off* error-handling state that shuts down the ECU's CAN communication. This attack is achieved by either physical manipulation of the bus [47, 46] or carefully synchronized and crafted transmissions [30]. However, these proposals either lack stealthiness against existing security proposals [17, 16, 20], require physical access [47, 46], or require strict control (e.g., synchronization) that cannot be achieved in practical remote settings [30].

For this kill-chain stage, we introduce a fundamentally different approach towards mounting shutdown attacks that, to the best of our knowledge, can evade all existing known defenses. Our attack is facilitated by architectural choices made to improve the integration and efficiency of automotive ECUs and their microcontroller units (MCUs). Modern MCUs typically integrate the CAN interface controller as an on-chip (CAN) peripheral in the same package. This design allows new inputs to the CAN peripheral to be accessible to the application-layer software via an application programming interface (API) and, thus, accessible to a remote adversary that successfully compromises an ECU.

We develop *CANnon*, a method to maliciously exploit one such input, namely the *peripheral clock gating* functionality. This particular API is accessible via software control in most modern automotive MCUs, often included as a valuable feature for performance optimization. We demonstrate how a remote software adversary can employ *CANnon* to utilize the CAN peripheral's clock to bypass the hardware-based CAN protocol compliance and manipulate the ECU output. This capability enables the adversary to inject *arbitrary* bits and signals (as compared to only being able to inject complete CAN-compliant frames) and gain the ability to precisely shape the signals on the CAN bus with bit-level accuracy. We demonstrate that this capability can be used to perform reliable and stealthy shutdown attacks. In other words, the modern MCU design has inadvertently strengthened the capabilities of a remote adversary, who is no longer constrained by CAN protocol compliance.

Our main insight here is the ability to *control* the peripheral's clock signal to then "pause" the ECU state in the middle of a transmission (or between state transitions). By exercising this control to selectively pause and resume an ECU's transmission, we can insert an arbitrary bit for a duration and at a time instance of our choice. This bit can be used to overwrite a victim's message and cause it to detect transmission errors. We also illustrate that the pattern of errors produced by *CANnon* is difficult to distinguish from legitimate errors on the CAN bus. Our fine control over malicious *bit* insertion (rather than message insertion) makes the detection of *CANnon* attacks difficult for currently proposed IDSes, as current techniques typically analyze *entire* messages for signs of malicious activity. Additionally, as *CANnon* does not involve spoofing IDs or overwriting the content of a message, even ID-based filtering at the data link layer [20] seems incapable of detecting our attack.[7] Preventing *CANnon*-based attacks require either architectural-level changes, such as isolation or removal of the clock control, or modifying existing approaches to specifically detect *CANnon*-like patterns.

In summary, we contribute the following:

- We introduce new methods to exploit the peripheral clock gating API of automotive MCUs to bypass hardware-based CAN protocol compliance and inject arbitrary bits on the bus. In contrast to previous work, we do not exploit diagnostic messages [41, 10, 42] and do not have tight synchronization requirements [30].
- We present three stealthy versions of *CANnon* and discuss modifications to make *CANnon* stealthy against future defenses.
- We illustrate both a basic denial-of-service (DoS) attack and a targeted victim shutdown attack atop two modern automotive MCUs used in passenger vehicles: the Microchip SAM V71 MCU and the STMicro SPC58 MCU. We validate the feasibility of this attack against a 2017 Ford Focus and a 2009 Toyota Prius and achieve a shutdown in less than 2ms.

4.3 Attack goals

Our remote adversary will likely target non-safety-critical ECUs (e.g., the head unit or navigation system), which often have remote wireless interfaces to handle multiple high-performance functions. As this adversary likely cannot gain direct compromise of a safety-critical ECU, the adversary will aim to utilize a compromised ECU to influence the functionality of a different

[7]Some recently proposed secure transceiver architectures use such filtering, but it is unclear from publicly available information whether they implement additional countermeasures. We have not evaluated any such products in the market to check their resistance against the *CANnon* attack.

(typically safety-critical) ECU in the vehicle without being detected by any deployed network security mechanisms (e.g., IDSes). One way to achieve this attack using the compromised ECU is to shut down a critical ECU and then disable its functions or impersonate it after the shutdown. In this stage, we focus on achieving a shutdown of a critical ECU without being detected by state-of-the-art network defenses, i.e., the adversary succeeds if the defense cannot detect an attack *prior* to the shutdown event. As we will demonstrate, the ability to reliably inject an arbitrary bit at an arbitrary time without being detected by vehicle defenses is sufficient to achieve these goals.

Thus, we effectively explore the possibility to construct a reliable remote bit insertion attack, which aims to shut down an ECU, operates as a software application, does not require access or changes to the physical CAN hardware, and deceives even state-of-the-art defenses. Furthermore, although several attacks outlined in Chapter 2 achieve similar goals, to the best of our knowledge, existing shutdown mechanisms cannot simultaneously be remote (performed only at the application layer), reliable (ability to consistently succeed), and stealthy (ability to deceive known defenses). The *CANnon* attack shows that the adversary model used by the industry has *changed* and that the attacker now has *new capabilities* that prior defenses did not consider. The notion of stealth is difficult to characterize, considering the rapid progress in defense mechanisms.

4.3.1 High-level attack insight

Contrast with prior invasive glitch attacks: Creating artificial clock glitches is a common technique to bypass security of MCUs during boot or verification by invasively driving the clock signal line to ground [42]. The idea behind such a technique is to create artificial transitions in the state machines implemented in hardware. As described in Chapter 2, the difficulty in injecting arbitrary bits is the CAN protocol enforcement by the CAN data link layer, i.e., the CAN controller. Thus, similar to the security logic above, the controller can be viewed as a hardware-based state machine that enforces CAN protocol compliance. Thus, we draw inspiration from the same direction of work but without requiring invasive physical access to the clock.

***CANnon* attack anatomy:** Any finite-state machine (FSM), e.g., the CAN protocol, implemented using sequential logic elements (flip-flops, registers, etc.) relies on the clock signal for state transitions and thus any output transmissions. Therefore, control of the clock signal can be used to force arbitrary deviations from the protocol. As an example, small changes in clock frequency would directly result in a change of the bit duration on the CAN bus.

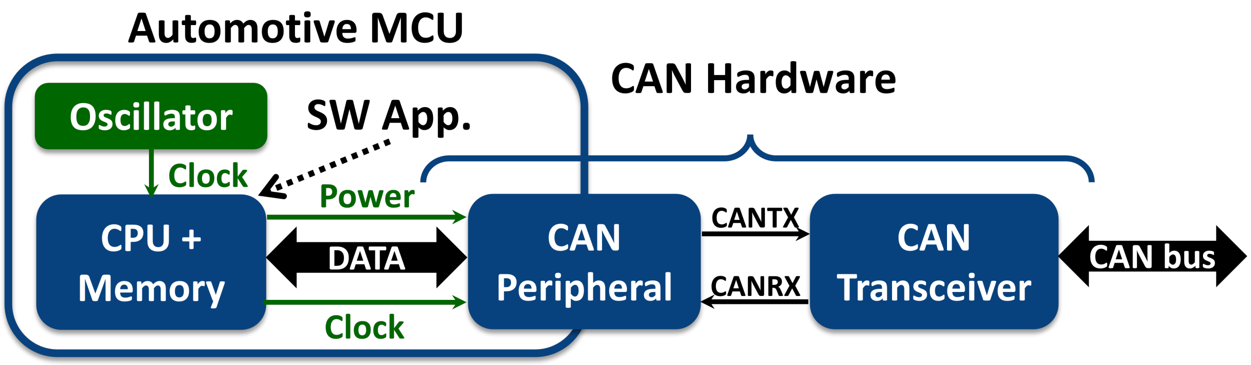


Figure 18: Modern ECU design with peripheral clock gating

Disconnect makes clock control possible: In an ideal design, the clock signal should not be accessible by a remote adversary. However, for modern ECUs, the MCU is a multi-chip module, where the CAN controller is integrated into the same package as the MCU and is now called a CAN peripheral. A simplified example of the modern ECU architecture is shown in Figure 18. Additionally, most modern MCU architectures implement power optimization in the form of *peripheral clock gating*. This low-power feature saves energy by shutting down any unwanted peripherals when they are not required, while allowing the main CPU and other critical functions in the MCU to still operate. As the CAN controller is typically attached as a peripheral to the MCU chip, there are controls exposed to cut off the CAN peripheral's clock.

To allow flexibility and control to low-level system designers, most MCUs provide the system designer a small software interface for the controls that allow clock cut-off. As we will demonstrate in our evaluation, clock control can be arbitrarily exercised during regular operations, which can also provide a remote adversary in control of the software with the same ability to control the CAN protocol FSM. This control effectively allows an adversary to *gate* the clock and *freeze* the protocol FSM, only to later restart the clock to resume the FSM. Thus, this new capability allows an adversary to arbitrarily manipulate the CAN protocol *without* modifying the hardware.

We note that, in most scenarios, cutting off the clock does not affect any data present in the sequential elements or the outputs of the logic gates. It simply prevents a change in the state of these elements. Also, without architectural changes, the notion of a *frozen state* or disabled clock cannot be recognized or corrected by the CAN controller. An alternative control in the form of power gating may also be available in certain chips, and we investigated exploiting such mechanisms. However, we find that disrupting the power simply resets the peripheral and its buffers/registers, causing the CAN FSM and output to be reset. Ultimately, we discover this attack vector in the driver code for the CAN peripheral. In hindsight, we realize that another factor that enabled our discovery of this vulnerability was our choice in experimental setup (detailed in our evaluation), which closely resembles the modern MCU architecture, whereas most prior research has continued to use the legacy architecture.

Overview of the attack: For any transmission, the CAN controller outputs the bits of the frame serially onto the transmit output pin (CANTX in Figure 18), where each new bit is triggered by a clock transition. The binary output of the controller is converted to a CAN-compatible analog value on the bus by the transceiver. Consider the case when the CAN controller is transmitting a dominant logical-0 bit. If the clock is disabled (paused) before the next bit, the CANTX output would continue to be logical-0 until the next clock edge. Thus, the node would continue to assert a dominant signal until the clock signal is resumed. This action effectively allows the transmission of a dominant bit of arbitrary duration. Now consider the opposite case when the CAN controller is transmitting a recessive logical-1 bit. If the clock is disabled, it would continue to assert a recessive value on the bus, i.e., no signal. The rest of the payload resumes transmission only when the clock signal is resumed. This action allows the transmission of the payload at an arbitrary time. Observe that the adversary exploits the controller's inability to sense the bus state when its clock is in the paused state. Thus, resuming the clock resumes the FSM from the point it was stopped, regardless of the current bus state or the controller's transmission output on the bus. This fact is key to disable the back-off arbitration control in CAN controllers and to transmit a signal at an arbitrary time.

4.4 Basic remote disruption attack

In what follows, we take a step-wise approach to increase the sophistication of our attack, ultimately demonstrating a controlled victim shutdown. In this section, we begin with a simple application of clock control to disrupt the entire network via a denial-of-service (DoS) attack. This basic disruption also highlights practical constraints that we must consider to design a reliable attack strategy. We note that this basic attack is easy to detect, and current hardware measures can sufficiently protect against it. However, the techniques we describe are the basis for precise and consecutive error injections required for a reliable targeted version of this shutdown attack.

Clock gating at application layer: The primary requirement for this attack is that the MCU must have control over the clock input for its peripherals, e.g., controllers for different protocols, such as CAN, Ethernet, FlexRay, etc. For the attack we present here, we choose a popular hardware device with a high-performance MCU built for networking applications: the Arduino Due board with an AT91SAM3X8EA 32-bit MCU operating at 84 MHz [78]. The Arduino Due offers many networking peripherals (e.g., CAN) and its source code (and CAN drivers) are well-documented, making it ideal for demonstrating our insights. In fact, we find that MCUs marketed as "high-performance" often include peripheral clock gating as a low-power feature available for the system designer (and thus a remote adversary).

Another requirement is that enabling/disabling the clock signal should not reset the peripheral circuitry or change values of its logic elements. Ideally, disabling the clock should only prevent the sequential elements from transitioning to a new state. This fact holds true for basic clock control mechanisms. For the APIs of the automotive MCUs we evaluate, we find the presence of multiple instructions that control the clock. Typically, for some of the commonly used APIs, MCU designers may implement additional check routines before/after a clock disable instruction to ensure error-free functioning, e.g., check status of transmission, etc. However, these procedures were only implemented for some of the available clock control instructions, and we find at least one instruction that offers a basic control mechanism.

To use the clock control, the adversary must identify which instructions enable an MCU's application to control peripheral clock signals. Typically, manufacturers provide basic driver code for initialization of several peripherals as part of their software development kit (SDK). In such cases, we can discover clock control instructions in the *drivers* for the CAN peripheral. Alternatively, in the event that all clock control instructions are not completely detailed, the reference/programming manuals for a given MCU often outline the steps required to control the peripheral clock and will provide the specific registers that control the clock gating. In the driver for the Arduino Due, we discover the instructions, `pmc_enable_periph_clk()` and `pmc_disable_periph_clk()`, to enable and disable the clock, respectively. These instructions appear prior to low-level configurations (e.g., memory allocation, buffer flushes, etc.). However, for another MCU popular in the automotive community, the STMicro SPC58, finding equivalent clock control instructions was more challenging as directly disabling the peripheral clock was not possible. Thus, we use its reference manual to identify specific registers that grants us a similar clock control capability in our evaluation.

Simple disruption attack: Recall that the CAN bus state is dominant if at least one ECU transmits a dominant bit. As a CAN frame consists of a series of dominant and recessive bits that follow a particular format, no valid information is conveyed from a single state held on the bus. Additionally, such a condition would result in continuous errors in the ECUs due to absence of stuff bits.

Thus, a basic attack we conceive is to *disrupt the bus* by holding the bus in the dominant state. This disruption would prevent any other ECU from transmitting, leading to a DoS of all ECUs in the network. An adversary could perform this action at a critical time (e.g., while the vehicle is in motion) and disrupt key vehicle functionality. For most vehicles, this attack would result in loss of control by the driver.

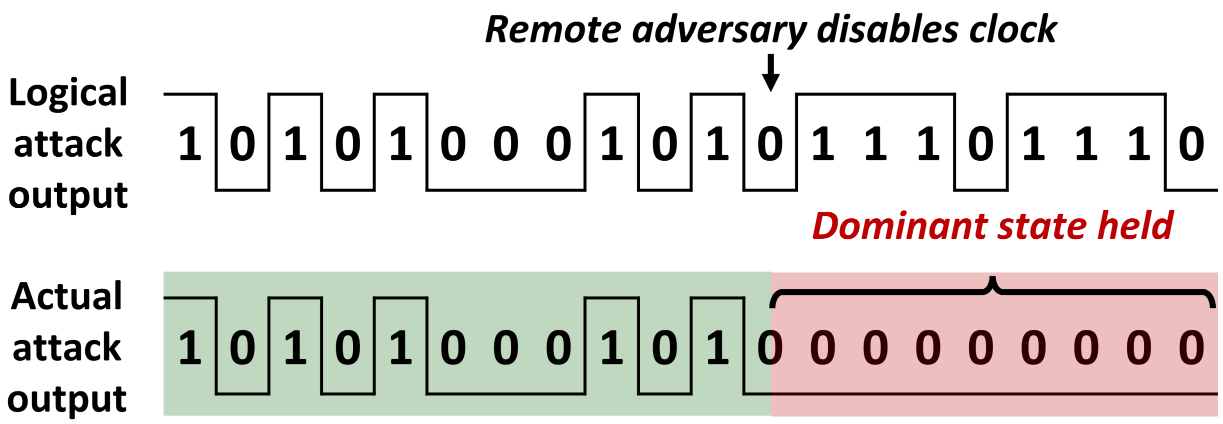


Figure 19: Holding dominant state disrupts the bus

Using clock control instructions, the adversary could easily achieve this attack by disabling the clock and *freezing* the CAN controller state when it transmits a dominant bit. To launch this attack, a basic method is to target the start-of-frame (SOF) bit:

- Send a message transmission request to the CAN peripheral with any priority and payload.
- Use a timer to delay for half a bit length for the given bus speed so the peripheral starts the transmission of the SOF bit.
- Pause the clock using the disable command to freeze the state of the CAN controller during the SOF bit.

If the bus was initially idle, this sequence would likely lead to the node continuing to hold the dominant state as depicted in Figure 19. However, there are several practical challenges evident from these basic steps. One critical challenge we encounter is the precise timing required to freeze the controller during the target SOF bit. In practice, the selected delay value only works if the bus was idle when the transmission request was sent and the frame immediately transmitted.

In a real scenario, the transmission may start much later (e.g., other bus traffic, scheduling delay, etc.). Even minor variations in the timer used to realize the delay period can cause an adversary to miss the SOF bit. Furthermore, any variation in the latency of the actual clock gating effect from the time that the instruction was issued can cause an adversary to miss the SOF bit.

Although there are practical constraints in this attack, the simplicity of this attack (as a result of our attack insight) affords an adversary a great deal of flexibility. For example, missing the SOF bit could be compensated for by using an ID of 0x0 and data payload of 0x0 (essentially a message of all zeros). Thus, freezing the controller during the arbitration or data payload field would also disrupt the bus. However, even this all-zero message has recessive bits due to bit stuffing when converted to a CAN frame. Thus, accidentally encountering those bits due to unreliable timing can cause the attack to fail.

This disruption attack is easy to prevent (if not already prevented) by most modern CAN buses. This disruption attack closely resembles a typical hardware fault encountered in a real CAN bus (i.e., bus held-dominant faults). Thus, several existing CAN transceivers implement built-in mechanisms to prevent the bus from holding a dominant state for a long period of time. This attack demonstrates the practical feasibility of using the clock control to launch an attack. This attack, though potentially dangerous, is highly obstructive for all nodes. It is still short of the goal of this work, which is to target a single ECU with high reliability and without being detected.

4.5 Reliable target victim shutdown

We now address some of the limitations discussed in the previous section to achieve a *reliable* attack that can *target a specific victim* ECU and quickly shut it down. We detail three variants of the *CANnon* attack and provide solutions to challenges observed in practical scenarios.

Reliable clock control: We previously illustrated the difficulty to ensure the clock is paused during a dominant bit. In general, an adversary with unreliable control of the clock cannot precisely ensure what state the controller outputs. Also, unlike the previous attack, a targeted attack usually requires overwriting specific bits of a victim message, thus requiring even more precision. One source of this unreliability is the variation in latency of the clock gating instructions, before the clock is actually paused. Another issue for this attack is that the adversary must track the state of the CAN bus and its own transmissions in order to target specific bits. However, when the CAN controller is in the frozen state, it does not have the ability to observe the CAN bus state. Without feedback, the adversary is virtually blind to the actual peripheral output while performing an attack. Thus, the adversary must keep track of which bit of a compromised ECU's frame it is transmitting at all times.

When the adversary calls a clock gating instruction (either enable or disable), we experimentally find that it takes up to two MCU clock cycles for the instruction to resume the peripheral's output. Thus, the adversary cannot reliably gate the clock near the edge of a bit transition of the attack message. A nonzero latency means that the adversary cannot ensure whether a gating instruction results in the output of the state before or after the state (bit) transition. This latency can thus influence the state of the bus that is held when the controller is frozen. Additionally, an adversary will need to make repeated calls to gating instructions within a single frame transmission by the compromised ECU. If the adversary loses precision in their clock control at any time, they could lose track of which bit the compromised ECU is currently transmitting.

Improving precision: To address the challenge of reliable clock control, the adversary can take advantage of the fact that the MCU's clock operates at a much higher frequency than the CAN peripheral's clock. We utilize the MCU's hardware-based timer, operating it at a frequency equal to the bus speed. This timer creates interrupts at the middle of each CAN bit, which allows us to track exactly which bit the compromised ECU is transmitting. Prior to starting the timer, the adversary must first detect when the compromised ECU attempts to send a frame; from this point, the adversary should delay half of a bit time before starting the timer interrupt. Our solution is to gate the clock as close to the *middle of a CAN bit*, giving the adversary maximum distance from bit transition edges. With an interrupt at the middle of each bit, the adversary can reliably track the bus state and control the clock with bit-level precision.

Insertion of a single bit: The precise clock control described so far can be used to insert a single bit on the bus. As described in the previous section, simply disabling the clock is not sufficient for the adversary to relinquish bus control. It must be ensured that the clock is disabled during a recessive transmission so that the adversary can continue its attack at a later time (recall that a recessive output does not influence the bus). Since the adversary only has clock control at the middle of a bit, the following steps are required to inject *a single dominant bit*, assuming the compromised ECU is currently paused at the recessive state: (1) enable clock a half-bit time before recessive-to-dominant edge, (2) wait one bit time to enter dominant bit, (3) wait another bit time to enter recessive bit, and (4) pause clock a half bit-time after dominant-to-recessive edge. Thus, the adversary must use such a pattern of bits within its payload, i.e., a dominant bit between two recessive bits.

However, this attack pattern introduces another unique challenge. As described earlier, the ECU reads bus state after each transition. Thus, if the adversary stops its attack during a dominant transmission by the victim, the compromised ECU will raise an error since it transmitted a recessive bit (a stopping requirement for the adversary) but observed a dominant transmission. This error will cause the attack ECU to reset its transmission so we must investigate methods to overcome this challenge as discussed below.

Causing an error on the victim: We now discuss how to exploit clock gating to induce just a single error on a victim. Our goal is to trick the victim into detecting an error in the transmission of its own frame, causing its transmit error counter to increase. To achieve this, the adversary must induce an error *after* the victim wins arbitration and becomes the sole transmitter. As detailed in Chapter 2, a node transmitting on the bus simultaneously checks the bus for any bit errors. Thus, the adversary can simply overwrite a victim's recessive bit with a dominant bit using the steps outlined in the previous section, tricking the victim into thinking it caused the error. To successfully achieve this, there are two practical challenges that the adversary must consider: (1) it must account for the victim response (i.e., error flag transmission), and (2) it should identify bits in the victim frame that can be reliably targeted.

Victim's error response: When the adversary overwrites a victim's recessive bit with a dominant bit, the victim will detect a bit error and immediately transmit an error frame. Depending on the state of the victim's error counter, this error frame can be a series of six dominant or recessive bits. However, as outlined above, an adversary cannot stop its attack during a victim's dominant transmission. Thus, an adversary cannot stop the attack if it expects the victim to transmit an active (dominant) error flag.

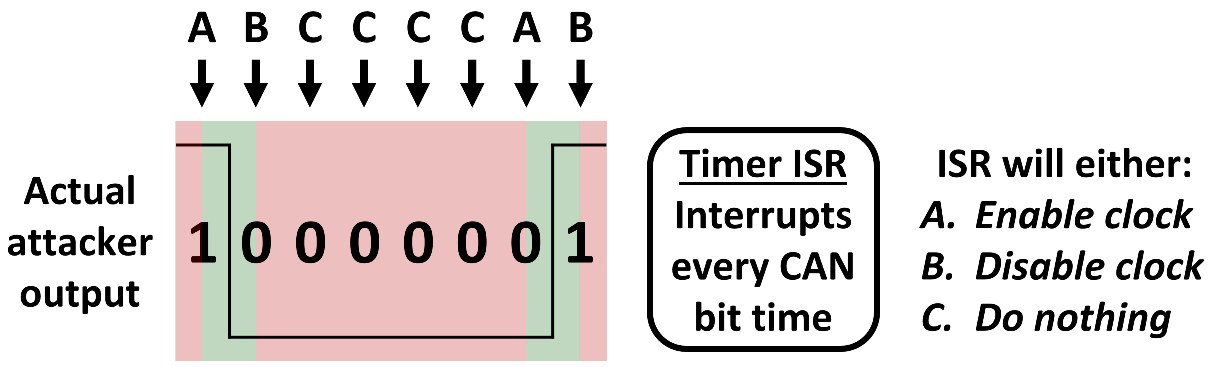


Figure 20: Use timer ISR to generate an error frame

To resolve this, the adversary can exploit their clock control to *expand* a single dominant bit into a series of six additional dominant bits, or an active-error flag. To generate an active-error flag from a single dominant bit, we perform four steps as depicted in Figure 20:

1. With clock paused on a recessive bit, the adversary resumes clock for one bit time (or until the next timer interrupt).
2. After the recessive-to-dominant edge, the adversary pauses clock so the compromised ECU holds dominant state.
3. After five timer interrupts, the adversary resumes clock.
4. The compromised ECU's output transitions from dominant to recessive, and the adversary pauses the clock at the next interrupt and is ready for the next attack.

By simultaneously transmitting the flag as the victim transmits its flag, both flags will overlap, and the compromised ECU's transition from dominant to recessive will occur when the bus state is recessive due to the recessive end of the error flag. This approach enables the attacker to maintain an error-free transmit state on the compromised ECU so it may prepare for the next error injection. In scenarios where there are multiple nodes on the bus, the length of the error frame may be longer and thus the dominant duration by the attacker should be adjusted accordingly.

Targeting victim frames: A challenge we face is determining which bit to overwrite during a victim frame. Assuming that the adversary can determine the starting point of the victim's transmission, identifying the location of general recessive bits may be difficult due to variations in the payload and stuff bits. Recall that, during the paused clock state, an attacker has no source of feedback from the bus. Thus, we must identify some invariant about the victim's transmission for the adversary to exploit. We borrow an insight from prior work to target the control fields, which often are static as data payloads do not change length [30]. Alternatively, the adversary could analyze several frames prior to attack and target bits in the data payload that remain static. However, as the stuff bits can vary, it is preferable to use the initial control bits for attack.

4.5.1 Shutting down victims with *CANnon*

We now stitch together the components described above to transition a victim into the bus-off state. To achieve the shutdown attack against a specific victim ECU, the adversary must cause enough errors to forcibly transition the victim into a bus-off state. The goal here is to produce an attack that operates as fast as possible. For now, we assume that victim transmissions are periodic, which is often the case according to prior work [31], and thus the period can be used to estimate a victim ECU's SOF bit transmission time. As depicted in Figure 21, the *CANnon* attack consists of two phases: a *loading* phase, where the attacker prepares the attack, and a *firing* phase, where the error frames are generated.

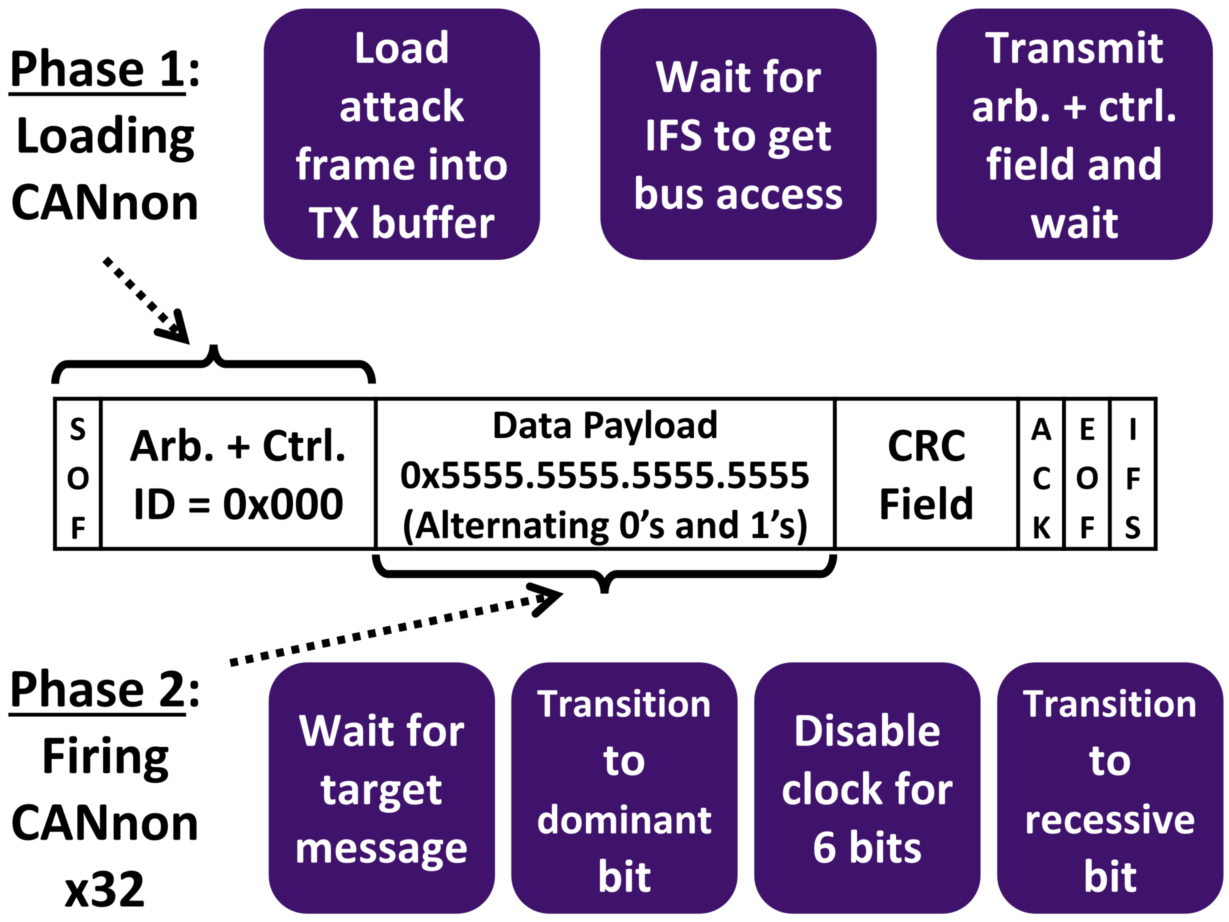

Phase 1:
Loading
CANnon
Load attack frame into TX buffer
Wait for IFS to get bus access
Transmit arb. + ctrl. field and wait
S O F
Arb. + Ctrl. ID = 0x000
Data Payload 0x5555.5555.5555.5555 (Alternating 0's and 1's)
CRC Field
A C K
E O F
I F S
Phase 2:
Firing
CANnon
x32
Wait for target message
Transition to dominant bit
Disable clock for 6 bits
Transition to recessive bit

Figure 21: *CANnon* shutdown workflow

Loading the *CANnon*: To be able to transmit any arbitrary signal on the bus, the CAN controller must first win arbitration. Since the adversary only controls the software and is unable to modify the CAN controller, the compromised ECU's FSM should be in the same state (sole arbitration winner) before the adversary can attempt to transmit arbitrary bits. Thus, in the loading phase, the adversary aims to trick the compromised ECU into thinking that it is allowed to transmit on the bus as preparation for the firing phase. To do this, the adversary loads the attack frame (of a specially selected ID and payload) into the CAN peripheral's transmit buffer and waits for the compromised ECU to win the bus. In this attack, the adversary waits for completion of the arbitration phase and transmission of the control bits, before pausing the clock during the first payload bit, which can be designed to be recessive. At this point, the adversary is ready to start the *firing phase* of the attack while waiting for victim messages to appear on the bus.

To ensure a quick transition into the firing phase, the adversary can set the arbitration ID of the attack frame to 0x0, giving it highest priority on the bus. This ID ensures that the compromised ECU wins control of the bus as soon as the bus is idle. Then, to transition a victim into bus-off, the adversary needs to inject a dominant bit 32 times using a single attack frame. Thus, the data payload should be set to contain 64 bits of data with a pattern of alternating 0's and 1's, or a payload of `0x5555.5555.5555.5555`. This payload gives the adversary 32 dominant bits to use for an attack and 32 recessive bits to temporarily pause the attack between victim transmissions. An additional benefit is that having a consistent pattern simplifies the logic for the adversary.

It should be noted that a different attack payload can still be utilized to achieve the same attack, albeit in a slightly sub-optimal manner. Any deviation from a payload of alternating dominant and recessive bits would require the attacker to reload another attack frame before shutting down the ECU.

Firing the *CANnon*: In the firing phase, the adversary utilizes the strategy described earlier to convert a single dominant bit into an active error flag, which will overwrite the recessive bits of the victim message. The adversary must wait for a victim message to appear on the network by waiting for its next periodic transmission that wins bus arbitration. The adversary then overwrites its selected recessive bit, causing the victim to detect an error and increment its error counter by 8. After detection of the error, the victim will immediately attempt to retransmit the failed frame. The adversary repeats this firing phase against 31 back-to-back victim retransmissions until the victim's error count reaches 256 (8x32) and enters the bus-off state. Thus, after the adversary causes an error in the first victim transmissions (using its period), targeting the retransmissions is significantly easier for the adversary.

4.5.2 Alternative *CANnon* implementations

Although the strategy described above is an efficient method to force the compromised CAN controller to transmit, we also describe alternative methods that achieve a shutdown attack using different parts of the CAN frame to highlight the flexibility an adversary has for the *CANnon* attack.

Firing with SOF bit: Instead of the above two-phase approach, imagine if the adversary could just skip to the firing phase. Our insight here is to use the SOF bit but with a different approach from our basic disruption attack. By stopping the clock right *before* a SOF transmission, the adversary can inject a dominant SOF bit during a victim's recessive bit. Since the SOF is only transmitted after a bus idle, the adversary can only transmit a SOF when it knows the bus is idle. Once bus idle is detected, the compromised CAN controller will load the registers to prepare for frame transmission. The adversary can pause the clock right when the transmit registers are loaded (experimentally, we find this to be two CAN bit times), effectively stopping the transmitter before it sends a SOF.

However, as the SOF is only a single bit, the error active flag from the victim will cause an error on the compromised ECU, forcing it to retransmit. Instead of avoiding this retransmission, the adversary can exploit it. The victim's error flag will cause the compromised ECU to think it simply lost arbitration. The adversary can then wait for a bus idle to occur and perform its attack again. Bus idle will not be observed until after the victim successfully retransmits so the adversary will need to target the periodic victim transmissions instead of its retransmissions from the loading/firing attack. While this attack is not as fast as the loading/firing attack, it does enable the *CANnon* attack on alternative MCU architectures as explained in our evaluation.

Firing with ACKs: Instead of using data frame transmissions to attack a victim ECU, the adversary could exploit another scenario where the compromised ECU transmits a dominant bit: the acknowledgement (ACK) slot. To acknowledge a correctly received data frame, an ECU will set a dominant bit during the ACK slot of the data frame. Our idea here is the adversary could pause the compromised ECU right before it transmits the ACK bit for a victim's frame (the bit before the ACK slot is a recessive CRC delimiter bit). Suppose the CAN peripheral offers a SOF bit interrupt, which we observe in two of our automotive MCUs [79, 78]. If the adversary knows when the victim frame transmission starts and can determine when the CRC delimiter bit occurs in the frame, the adversary can pause the clock before the ACK slot and resume the clock just a few bit times later during the EOF, causing an error on the victim. The challenge here is that the adversary must precisely predict when an ACK will occur and the number of bits in the victim frame. Thus, victim frames that contain static or predictable data make an ideal target.

4.5.3 Practical challenges

We now discuss approaches to solving two practical challenges we encounter when launching *CANnon* against real vehicles in our evaluation, one of which is a new capability resulting from the peripheral clock gating vulnerability.

Period deviations in victim frames: Up to now, we make the assumption that victim frame transmissions will be periodic. However, in practice, our work on *CANvas* has found that period deviation is nonzero, which makes it difficult for the adversary to predict victim transmission times and thus perform the shutdown attack. Using insights from prior work [30], we could estimate when a victim message will initially appear on the bus. However, these insights relied on other messages in the network that would transmit immediately before the victim message, which is not always guaranteed. Likewise, even considering these circumstances, this approach has been found to not be reliable [31].

We introduce a new capability that permits an adversary to *guarantee* when a victim message appears on the CAN bus. We first revisit an observation made when launching the basic disruption attack during tests on real vehicles. When the compromised ECU holds a dominant state, all other ECUs will queue their frames waiting to transmit during bus idle. Upon releasing this dominant state, all transmitting ECUs will attempt to clear their queues. We find that these queued frames appear on the bus in a pre-defined order: by their arbitration ID. Our insight here is to determine which messages should arrive in a given range of time prior to launching our attack. By holding the dominant state for this range of time,[8] we can predict the ordering of messages and thus predict the start of the victim transmission.

Interruptions by higher-priority messages: Another practical challenge we encounter when launching *CANnon* against real vehicles is that higher-priority messages can interrupt the attack. If the adversary targets a victim frame with a low priority, we find that higher-priority messages can interrupt the repeated retransmissions by the victim. As the adversary expects the victim retransmissions to occur back-to-back, these interruptions can cause the attack to fail by causing collateral damage against unintended victims. Thus, the adversary could use the *CANvas* network mapper to identify all source IDs of a victim ECU and simply select the highest-priority message, minimizing the chance of interruption by a higher-priority message. Additionally, our work on *CANvas* also finds that safety-critical ECUs tend to transmit higher-priority frames so our adversary is already incentivized to target higher-priority frames.

4.6 Evaluation

In this section, we demonstrate *CANnon* using two automotive MCUs found in modern vehicles and launch shutdown attacks against a variety of targets, including two real vehicles. We also detail experiments to highlight the reliability and stealth of *CANnon*.

Experimental setup: To demonstrate the significance of this attack, we launch *CANnon* from automotive MCUs used by modern vehicles, and we target real ECUs from two real vehicles. In this work, we do not explicitly show the ability to compromise an in-vehicle ECU remotely as this has been the focus of a large number of papers [10, 22, 35, 36, 37]. Rather, we build our attack on the assumption that these existing techniques would be successful in remotely compromising the software of automotive ECUs.

One of the key factors that enabled the discovery of this vulnerability was our choice of

[8]Where prior work required injecting a message to guarantee transmission time of a victim, we can simply disrupt the bus to "simulate" an injected message.

experimental setup, which is likely why, to the best of our knowledge, this incidence has not been studied before. In this work, we initially used the Arduino Due board, which closely resembles the capabilities of modern automotive MCUs. However, if we look at prior work in the field, we find widespread use of the legacy design of automotive ECUs, namely an Arduino Uno board with a standalone controller [30, 15, 16, 43, 47, 17, 21, 80]. Thus, as a result of *our choice of experimental setup*, none of these prior works could have identified the *CANnon* vulnerability; where the industry moved to a modern design, prior research has continued to use the legacy design.

Automotive MCUs: In addition to the Arduino, we test *CANnon* on evaluation boards for two automotive MCUs from Microchip and STMicro (commonly known as ST). These boards will serve as the compromised in-vehicle ECUs as they are used in modern production vehicles. In fact, STMicro is one of the top five semiconductor suppliers for the automotive industry [81], and its MCUs are likely to be in many modern vehicles. The features and architectures they offer are likely generalizable to other automotive MCUs as they are both marketed as high-performance networking MCUs, which are two key features we identify in our simple disruption attack. While we do not evaluate boards from every MCU supplier, we find multiple references to software APIs for peripheral clock gating in reference manuals and market reports [82, 83, 84, 85, 86].

Specifically, we evaluate: (1) the Microchip SAM V71 Xplained Ultra board, which uses an ATSAMV71Q21 32-bit MCU operating at 150 MHz and is designed for in-vehicle infotainment connectivity [87, 79], and (2) the STMicro SPC58EC Discovery board, which uses an SPC58EC80E5 32-bit MCU operating at 180MHz and is designed for automotive general-purpose applications [88, 89]. It is likely that other MCUs in the same family (i.e., SAM V MCUs and ST SPC5 MCUs) share the same peripheral clock gating vulnerability as demonstrated by similarities within an MCU family's reference manuals [79, 78]. Consequently, the Arduino Due board identified in our simple disruption attack uses an AT91SAM3X8EA 32-bit MCU operating at 84 MHz from an older series of the same SAM family [78].

For the SPC58 MCU, we encountered a challenge in finding a clock enable/disable function. All clock functions *requested* the peripheral to essentially give the MCU permission to disable the peripheral's clock. Upon receiving the request, the peripheral waits for all transmission operations to stop before the MCU can disable its clock. However, we found an alternative approach to directly control the clock on the SPC58 that bypasses this request procedure. In fact, this alternative contradicts the expected implementation as described in the SPC58's reference manual [89]. The SPC58 utilizes operating modes that change the configurations for the MCU. In particular, we focus on two modes: the DRUN mode, which permits all peripherals to run normally, and the SAFE mode, which stops the operation of all active peripherals. We find that a transition to DRUN is equivalent to enabling the CAN peripheral's clock and a transition to SAFE effectively disables the peripheral's clock without permission from the peripheral itself. A limitation introduced here is that a clock enable could not occur soon after a clock disable,[9] but the SOF-based attack from our targeted victim attack successfully works for the SPC58.

Real vehicle testbed: Additionally, we demonstrate *CANnon* against two real vehicles: a 2009 Toyota Prius and a 2017 Ford Focus. We connect to the CAN bus by accessing the bus via the vehicle's On-Board Diagnostics (OBD) port to emulate a remotely-compromised ECU. We also identify the mapping of arbitration IDs to source ECUs using details from our *CANvas* network mapper. We only launch the *CANnon* attack against vehicles while they are parked to avoid any potential safety concerns. Note that these vehicles have their engine running with all ECUs actively transmitting onto the network. As detailed in our *CANvas* work, vehicles tend to transmit mostly periodic messages, and we find that these transmissions start when accessories are turned on. Even if the vehicle is taken on a drive with the engine on, only the data payloads change rather than periodic transmission rates. We also launch *CANnon* against an Arduino Due, a PeakCAN USB device, and a 2012 Ford Focus powertrain ECU in a variety of synthetic network setups.

[9]The transition to SAFE mode (or effectively disabling the clock) includes several processes that must complete for safety reasons before the peripheral clocks can be enabled.

4.6.1 *CANnon* against real vehicles

Basic disruption on real vehicles: We launch the basic disruption attack against both real vehicles using the SAM V71 and SPC58 evaluation boards. As discussed in our basic disruption attack, ECUs often implement a time-out feature that prevents a CAN transceiver from holding the dominant state for an extended period of time. We experimentally find that we can maintain up to 1ms of dominant state on the bus with at least 4μs of recessive in-between on both vehicles. We find that this attack prevents ECUs from communicating on the bus and will trigger malfunction lights on the instrument panel and even diagnostic codes that indicate loss of ECU communication.

Powertrain ECU shutdown in 2017 Focus: We demonstrate a shutdown attack with the V71 MCU using the loading/firing attack in our targeted victim attack. The powertrain ECU transmits several arbitration IDs, but we select the highest-priority ID using the *CANvas* network mapper. In our pre-analysis of the victim transmission times, we find that a majority of the powertrain ECU's IDs will transmit back-to-back. With our technique for guaranteeing transmission times, we hold the dominant bit when we expect the victim to appear (for approximately 50μs). Upon release of the dominant bit, the target victim frame will be the first frame to transmit and, thus, we launch our firing phase on that frame. We target the control field and perform this attack 32 times, allowing us to shut down the powertrain ECU in about 2ms. Although the powertrain ECU does auto-recover, the ability to shut down the ECU quickly demonstrates the speed of our attack.

Power steering ECU shutdown in 2009 Prius: We demonstrate a shutdown attack with the SPC58 MCU using the SOF-based attack in our targeted victim attack as the SPC58 cannot enable the clock immediately after disabling it. The target victim is a power steering ECU that transmits three IDs: 0x262, 0x4C8, and 0x521. We choose the ID with the smallest period (0x262 with period of 20ms) and find that its period deviation is quite small using the *CANvas* network mapper. As the SOF approach requires a successful transmission between each attack, this shutdown is significantly longer since we do not target retransmissions. We shut down the power steering ECU after 700ms, and we find that it remains permanently offline.

Attack reliability: One important aspect of a reliable attack is repeatability. We envision an adversary who purchases the same MCU that the compromised ECU uses as preparation for their remote exploit and shutdown attack. After tuning attack parameters to the specific MCU (e.g., number of MCU cycles prior to SOF transmission), the adversary hopes that the tuned parameters will be similar to that of the real victim MCU. We find that properly tuned attack code across multiple copies of our test MCUs over a few months could repeatedly produce the same output to the bus. We attribute this success to the strict specifications that ECU hardware must follow in the manufacturing stage.

We now compare the reliability of *CANnon* using the hardware timer interrupt centered on each CAN bit versus manually counting MCU clock cycles. In this experiment, we use the Microchip and Arduino Due boards to transmit active error frames at repeated intervals. We transmit these frames against an Arduino Due victim that sends its frames every 10ms with zero deviation in the period. Using a hardware timer to launch our attack, we find that both the Microchip and Arduino Due boards can shut down the victim 100% of the time. However, if we try to perform the active frame transmissions by manually keeping count of MCU clock cycles, we only achieve the attack 10% of the time due to variations discussed in our targeted victim attack.

We also compare the reliability to guarantee victim transmission time versus prior work that overwrites messages using injected messages to predict victim transmission [30]. Here, we use the Arduino Due board to target three different victims: (1) another Arduino Due, (2) a Peak-CAN device, and (3) a 2012 Ford Focus powertrain ECU. Using our method, we can achieve a shutdown of all three victims using all three of our MCUs 100% of the time. However, using prior work to perform the message overwrite attack, we only succeed for the Arduino Due and Peak-CAN device. On the powertrain ECU, we cannot achieve even a single success as its transmissions exhibit significant period deviation.

Stealth analysis: We now compare the stealth of *CANnon* versus the state-of-the-art message overwrite attack [30]. We construct three simple detection methods at each layer of the CAN stack based on existing defenses.[10] The goal of either shutdown attacker is to achieve a victim shutdown without the detection method alerting *prior* to the shutdown itself. Our experimental setup involves three Arduino Due boards: (1) the victim ECU, (2) the detection ECU, and (3) the compromised ECU. The detection ECU also transmits its own messages to simulate other traffic on the bus. We perform each test 1,000 times, and we operate all tests at 500Mbps, use a shared 12V supply to power the boards, and observe the bus traffic using a logic analyzer.

For all of the experiments below, we follow the configuration of prior work [30]: the victim ECU transmits ID 0x11 every 10ms, the detection ECU transmits ID 0x7 and 0x9 every 10ms, and the compromised ECU monitors the bus and attempts to attack. To simulate noise from a real vehicle, we intentionally set the deviation of ID 0x11 to 0.15ms as the best-case minimum deviation found by our work on *CANvas*. For all experiments with the overwrite attack, the compromised ECU injects ID 0x9 around the expected transmission time of 0x11 to set up their attack.

Versus timing-based IDS: We first test the overwrite attack and *CANnon* against a timing-based IDS that alerts if frames transmit outside of their expected period. Timing-based IDSes also include ML-based [56] and traffic anomaly methods [57] as they analyze timestamps to detect illegitimate transmissions. We set the detection threshold for period deviation to be 10% (e.g., 1ms for a 10ms period) following prior work [48]. We program our detection ECU to measure the inter-arrival time between frames for a given ID and alert if the measured time exceeds 10% of the expected period. For *CANnon*, the compromised ECU attacks using the data payload and employs the *dominant-hold* technique identified in our targeted attack to guarantee victim transmission time. Out of 1,000 attempts, we find that our detection ECU alerts to *every* attempt by

[10]We do not demonstrate *CANnon* against complete implementations of existing defenses, which monitor only entire CAN messages or frames, as they are ineffective by construction.

the overwrite attack but does not alert to any of the *CANnon* attacks. *CANnon* only needs to hold the dominant state for 0.15ms *once* to guarantee the first victim transmission and cause an error. The overwrite attack injects new messages onto the network, exceeding the expected deviation threshold. *CANnon* achieves a shutdown in just 2ms before the next transmission should occur.[11]

Versus a "secure transceiver:" As secure transceivers are not currently in production, we modify the detection ECU to act as the secure transceiver. It will read each logical bit transmitted and received by the compromised ECU by directly connecting between the MCU's CAN peripheral and the CAN transceiver following prior work [31]. If an ECU sends an illegitimate arbitration ID, it will produce an alert in real-time immediately after the arbitration field transmits. For *CANnon*, the compromised ECU attacks via the SOF bit method as the secure transceiver could detect the data payload attack.[12] Out of 1,000 attempts, we find that our secure transceiver alerts to *every* attempt by the overwrite attack but does not alert to any of the *CANnon* attacks. *CANnon* only injects a SOF bit as its attack and does not transmit any arbitration ID, while the additional message transmissions in the overwrite attack cause our secure transceiver to alert immediately.

Versus a frame-level voltage IDS: Following observations from prior work [16, 17, 50], we modify the detection ECU to directly measure the CAN bus voltages to detect an attack. The CAN medium is a differential pair called CAN low and CAN high that typically exhibit around 1.5 and 3.5 voltages for a dominant bit, respectively (recessive state causes both to exhibit 2.5 volts). The key insight from prior work is to measure the voltage of the dominant bits throughout an entire frame. With the message overwrite attack [30], the start of an overwritten frame has two transmitters and ends with a single transmitter (i.e., the compromised ECU). Thus, the attack exhibits a larger differential at the start and a smaller differential at the end of an overwritten frame. We build a voltage IDS that alerts if the dominant bits exhibit a sudden drop in dominant

[11]This fast ability to shutdown could act as a useful stepping-stone to future work on masquerade attacks.

[12]*CANnon* could technically use any arbitration ID (even a legitimate ID), but we assume that the adversary wants to use ID 0x0 to minimize wait for bus idle.

differential voltage during a single frame. Out of 1,000 attempts, we find that our IDS alerts to *every* attempt by the overwrite attack but does not alert to any of the *CANnon* attacks. *CANnon* only injects a single error flag in the *middle* of a frame and, thus, this approach to voltage IDS does not detect our attack.

4.7 Stealth against network defenses

Next, we detail how *CANnon* can deceive several state-of-the-art defenses and even some potential *CANnon*-aware designs as a demonstration of this critical vulnerability.

4.7.1 Deceiving state-of-the-art defenses

Many approaches exist that can defend against shutdown attacks. We group these defenses into three classes based on the layer in the CAN communication stack they operate on.

Defenses at application layer: Many IDSes are software applications, limited to reading messages passed up the communication stack by CAN hardware. These run on any ECU and do not require special hardware, making them an attractive solution. For instance, they can use statistical techniques based on message timings and content [43, 15, 48, 49]. A recent U.S. agency report discusses how companies working closely with automakers have access to proprietary information on the expected content of CAN messages, enhancing their ability to create application-layer defenses [29]. Another class of IDS that makes use of this proprietary information are machine learning [56] and traffic anomaly IDSes [57], which analyze message timing and payload to detect an attack.

Application-layer IDSes can detect both the diagnostic shutdown command and message overwrite attacks as they require transmitting additional CAN frames on the bus. As such, any application-layer defenses that measure message timing or content cannot detect our attack since we do not transmit entire CAN frames or significantly disrupt valid transmitted frames. *CANnon* operates quickly and can shutdown ECUs in just a couple milliseconds (well under the minimum period observed by our work on *CANvas*) as demonstrated in our *CANnon* evaluation.

Defenses at data link layer: Recent industry solutions propose secure CAN transceivers that operate at the data link layer [20]. These transceivers can *prevent* a compromised ECU from attacking a CAN bus by either: (1) invalidating frames with spoofed CAN IDs, (2) invalidating frames that are overwritten by a compromised ECU, and (3) preventing attacks that flood the bus with frame transmissions. Attacks that require physical access are outside their scope.

These transceivers are a promising approach to defending against a diagnostic shutdown attack and message overwrite attack as the transceivers would destroy any illegitimate frames based on their IDs. As the loading phase in our loading/firing attack transmits a specific arbitration ID (0x0), these transceivers would also detect an illegitimate ID from the compromised ECU and raise an alarm. However, the two attack alternatives (SOF and ACK attacks) do not produce an arbitration ID and could not be detected by pure ID-based filtering concepts as demonstrated in our evaluation.

Defenses at physical layer: Another approach for IDSes is to directly access the physical layer (e.g., measuring bus voltages). These defenses detect sudden changes over a series of CAN frames (or even a single frame) by creating a profile of the expected voltages [16, 17, 50]. These works find that each ECU applies a unique voltage that is measurable across an entire CAN frame. If an illegitimate transmitter attempts to spoof a victim's message, the voltage measured across the frame could identify a potential attack.

This approach can detect the message overwrite attack because a single frame will start with two simultaneous transmitters followed by only the overwriting compromised ECU; a distinctive change in voltage for the rest of the frame indicates an attack. However, in regard to physical-layer defenses that measure voltage, *CANnon* does not require overwriting a frame from the SOF onwards and, thus, prior work would not detect a sudden change in the voltage from the start of a single data frame [50] as demonstrated in our evaluation.

4.7.2 Deceiving *CANnon*-aware defenses

We now discuss how *CANnon* could remain stealthy against even future *CANnon*-aware defenses. We discuss defenses that might seem appealing at a glance, but we will show that this attack will likely require architectural countermeasures.

Tracking error interrupts at application layer: Up to now, we have discussed how application-layer defenses that only monitor messages do not detect *CANnon*. However, there is another source of signals from the lower CAN stack layers: error interrupts. We envision a *CANnon*-aware defense that uses these interrupts to identify potentially malicious sources of error. This defense tracks errors based on their frequency and for which messages they occur during in an attempt to find a pattern representative of a shutdown attack. Existing work can detect when a shutdown occurs by tracking error flags [21], but it cannot determine if the errors were caused maliciously or by legitimate bus faults. We now discuss a couple modifications that similar work could implement to detect a malicious attack. We also discuss how our adversary can thwart those efforts by making it challenging for defenses to detect *CANnon* while maintaining a low false positive rate:

1. *Tracking number of errors per ID:* One potential defense is to track the number of errors that occur when a particular message ID is transmitted. However, our adversary could use the *CANvas* network mapper to identify all source IDs from an ECU by simply monitoring the bus and tracking message timestamps. Our adversary could then target all IDs from a victim ECU, making an error seem consistent across all transmissions and difficult to differentiate from a legitimate fault.

2. *Checking for multiple errors in short time:* Another defense is to check for multiple errors in a short amount of time, which is an invariant of prior work [30]. While the loading/firing attack causes multiple errors in a matter of milliseconds, an adversary can extend this attack over a longer period of time. An active error flag will increment the victim error counter by eight; to recover from an error, a successful transmission from a victim will

decrement the error counter by one. Our adversary could launch an error flag for one of every seven successful transmissions from a victim, giving us an effective increase of one for the transmit error count. By repeating this attack 256 times, the adversary could allow up to 1792 successful transmissions by a victim and still succeed in a shutdown.

4.8 Countermeasures

We now identify countermeasures that target the crux of the vulnerability, which will likely require additional hardware changes. As illustrated above, *CANnon*-based attacks are stealthy against existing security methods. Here, we describe some directions for potential countermeasures. Since the attack relies on two broad ideas, namely clock control and error exploitation, the countermeasures described can be seen to prevent one of these problems, i.e., prevent clock control or detect specific error patterns and/or error transmitter patterns.

Detecting bit-wise voltage spikes: Overwriting a message causes a sudden voltage change in the dominant bit. Thus, one approach to detect such an attack is tracking per-bit voltages at the physical layer. Changes in the middle of message transmissions could indicate potential adversary activity. However, since random faults or genuine errors flags can cause similar behavior, such a method would require additional identification of patterns in the voltage changes, e.g., behavior periodicity. Some recent work that uses transition characteristics for network fingerprinting [80] could be modified in this direction.

Forced clear of transmit buffers: As observed in our attack insight, the ability to resume a message transmission is a key factor for successfully exploiting the controller. Thus, the attack can simply be prevented by disabling such behavior, i.e., resetting/clearing all buffers upon clock gating. Such a countermeasure allows the flexibility of being deployed at either the hardware or software level. If hardware changes are permitted, this approach can be achieved by designing reset logic based on the clock signal. In software, this approach can be achieved by flushing the peripheral transmit buffers upon clock stop. A modification of this idea for safety is present in

SPC58, whereby a clock stop request is completed based on the feedback from the CAN peripheral.

On-chip power analysis: The automotive industry takes another approach to protecting their ECUs from remote adversaries: host-based IDSes [29]. One host-based detection method for *CANnon* could be a separate secure chip that monitors the power usage of the MCU. Since disabling the peripheral clock induces a drop in power, a host-based IDS could detect potentially malicious actions. This approach should operate outside of the MCU and could include logic to identify when power drops are *not* expected (e.g., while in motion, while vehicle not asleep, etc.).

Removal of CAN peripheral clock gating: The main feature that enables *CANnon* in modern MCUs is peripheral clock gating. Rather than offering a peripheral for CAN, modern MCUs could simply utilize a separate always-on clock domain for the CAN peripheral or require standalone CAN controllers, which receive a clock signal from a separate oscillator. Assuming the other peripherals do not share this vulnerability, they could remain unchanged by removing clock gating for just CAN.

4.9 Summary

In this chapter, we presented *CANnon*, a disruption attack that can impact the transmissions of other ECUs on the network and even shut down a specific target ECU. *CANnon* leverages a new feature (i.e., peripheral clock gating) that can be exploited to permit a remote adversary to impact physical CAN signals even though a CAN controller is in place to enforce CAN format on the bus. By exploiting this disconnect, *CANnon* can precisely disrupt bus transmissions in real-time and even shut down powertrain ECUs on real vehicles. As a result, we can enable our adversary to control when messages appear on the bus and the timing of their transmissions in the next stage of this kill-chain. We also propose countermeasures that target this disconnect by detecting malicious use of this new feature or by simply removing this feature altogether.

5 *CANdid*: Remote Authentication Bypass on Automotive Control Units

In this chapter, we discuss our contributions to the authentication bypass stage of our attack kill chain. The contributions of the thesis in this stage are:

- **Kill-chain goal:** In this stage, we aim to develop a bypass authentication attack that can gain access to privileged commands on a victim ECU, such as writing new code to memory. With this ability, the remote adversary can now control a second safety-critical ECU (one without direct remote interfaces) in the victim vehicle to launch advanced attacks. We also need to demonstrate a proof-of-concept of this final goal so we aim to show that we can upload code to a powertrain ECU using our kill-chain.
- **Disconnect to exploit:** We find that the UDS implementation on ECUs can allow a remote adversary to request a hard reset without being authenticated. As an ECU's source of randomness for the challenge-response authentication is based on processor uptime (or time since last hard reset), the adversary now has control of when the hard reset occurs and thus can control what challenge the ECU produces, enabling a capture-replay attack.
- **Potential countermeasures:** The key vulnerability insight here is that the adversary can influence and even control the source of randomness. There are two approaches to countermeasures here: either detect an unauthorized tester (i.e., a compromised ECU) or prevent the attack by requiring authentication before hard reset requests or using a better source of randomness.

5.1 Goals for this kill-chain stage

With the ability to disrupt the CAN communications of both the target ECU and other in-vehicle ECUs, our attacker can proceed to the third stage of the kill-chain. Our attacker now aims to authenticate with the target safety-critical ECU, specifically for access that permits reprogramming the target's software. The UDS SecurityAccess service as detailed in Chapter 2 permits authorized users to gain programming access for a given ECU if they can pass the appropriate challenge-response authentication. Our attacker must consider how the automotive industry has both adapted to the current threat landscape and updated how this authentication is implemented. Here, we face better implementations of UDS SecurityAccess compared to implementations from prior work [7, 51]. The challenge is random to prevent brute-force and replay attacks, and we must assume that gaining access to secret keys is impractical as automakers may not place them on diagnostic testers. Thus, prior work that reverse-engineers keys from either ECUs or diagnostic testers would not succeed [10, 52]. In addition to this, most UDS implementations already block further attempts after a handful of failed authentication attempts. We imagine that state-of-the-art IDSes will use this mechanism to detect an adversary. As a result, we need an attack that can still bypass authentication without access to secret keys and do so with fewer than a handful attempts.

5.2 Stage overview and contributions

The landmark work on remote attacks by Miller et al. found that firmware on safety-critical ECUs will ignore critical attack messages (e.g., disabling the brakes) when above a low speed (e.g., 5-10 mph), thus severely limiting the real impact of their malicious message injection attacks [10]. Rather than relying on CAN message injection, an adversary that could instead *reprogram* a vehicle's engine ECU (or any other safety-critical ECU) could significantly strengthen an adversary's capabilities. With control of a safety-critical ECU's firmware, such an adversary could now access all of the ECU's inputs (from sensors, e.g., throttle position) and outputs (to actuators, e.g., fuel injectors) and could bypass any firmware constraints. For example, instead of only succeeding to disable the brakes at a low speed as seen in prior work [10], this adversary could reprogram the brake ECU and then disable the brakes at any speed by simply removing or bypassing safety-related program code.

The typical approach to legitimately reprogram an ECU over the CAN bus is to employ the Unified Diagnostic Services (UDS) protocol, an application-layer protocol that operates over a physical-layer network (e.g., CAN) and provides a variety of services to a dealership technician [73, 72]. Here, a technician will connect to a target ECU over the UDS protocol with a physically-connected proprietary diagnostic tester [34]. Then, to update an ECU's firmware for recalls, service, and upgrades, the technician must authenticate itself with the target ECU by passing a challenge-response authentication through the UDS SecurityAccess service. With the target ECU's secret key and encryption algorithm stored on the diagnostic tester, the technician can legitimately pass this authentication and then successfully download new code to the ECU.

To demonstrate an adversary's ability to maliciously reprogram an ECU over the CAN bus, researchers have identified several weaknesses in legacy implementations of UDS SecurityAccess, which checks if a user is authorized to access privileged commands. Prior work identified the use of fixed challenges, which permit brute-force attacks that simply try all possible responses , and the use of obvious encryption algorithms, which can be easily guessed (e.g., simple XOR with fixed value) [7, 51]. For more complex designs, researchers have reverse-engineered diagnostic testers [10] and ECUs [52] to extract both secret keys and encryption algorithms. Unfortunately, a common conclusion among these works is that keys and algorithms were shared across all vehicles of the same make and model. In other words, reverse-engineering the secrets for just one vehicle instance meant an adversary could ultimately target thousands of victims in the field using the same secrets.

With respect to the growing priority of automotive security, the modern implementation of SecurityAccess has evolved to address the weaknesses of legacy designs. Vehicles (especially fleets) can implement key diversification (i.e., a unique key per ECU instance), and even perform key rotation to change keys on a repeat basis, greatly reducing an adversary's ability to target multiple vehicles [52]. Likewise, these secret keys can be made accessible only to authorized technicians through the Original Equipment Manufacturer (OEM) rather than loading them onto diagnostic testers, preventing unauthorized parties from reverse-engineering these keys. Finally,

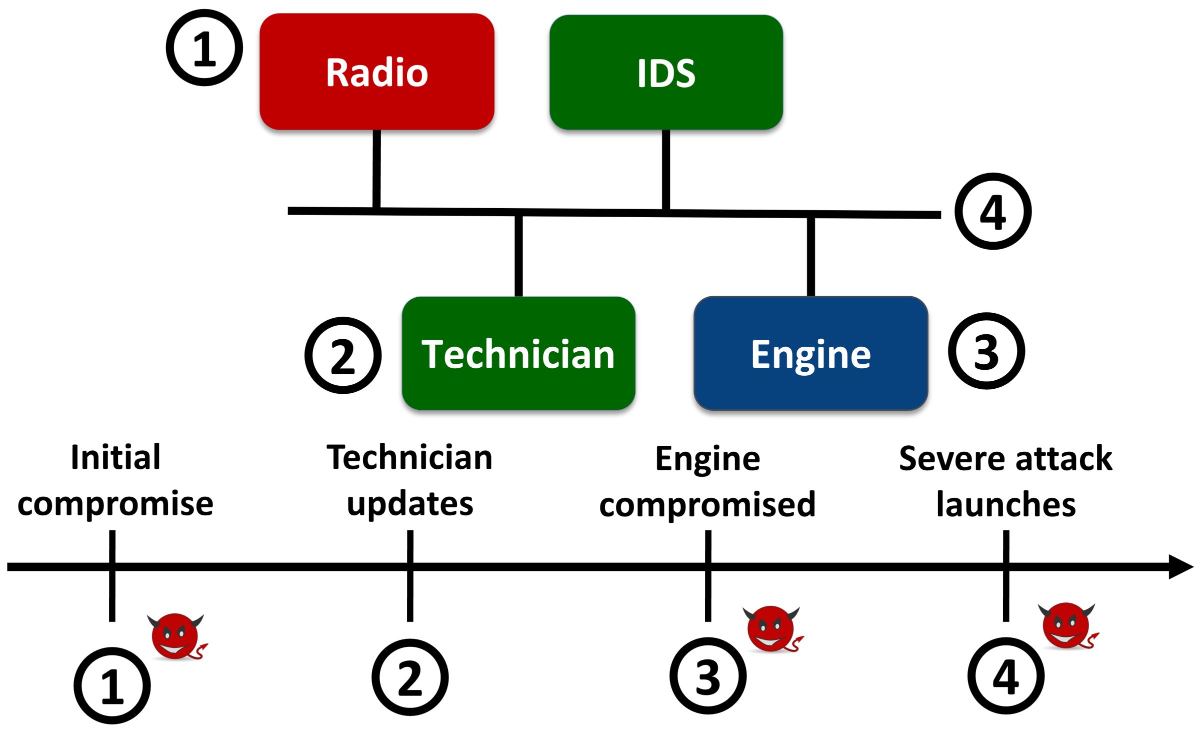


Figure 22: Timeline for authentication bypass via capture-replay

modern implementations of SecurityAccess use a psuedorandom number generator (PRNG) to seed the challenge. Here, an attempt to brute-force the response would be difficult and could be easily detected by a software-level CAN intrusion detection system (IDS).

In this stage, we investigate if a compromised ECU controlled by a remote adversary could act as a stepping-stone to reprogram a victim ECU, likely a safety-critical one, without being detected. We envision an adversary could successfully authenticate against this safety-critical victim ECU's modern SecurityAccess implementation without numerous failed attempts, which could trigger an alert by a CAN bus IDS. The adversary should not generate any obvious or known traffic indicative of a brute-force attack, and it should be capable of providing the correct response to a challenge sent from the victim ECU without prior knowledge of the target ECU's secret key. If successful, such an adversary in control of a safety-critical ECU could send signals directly to critical actuators without needing any communication over the CAN bus, thus enabling a very powerful attack.

We develop *CANdid*, a stepping-stone attack where an adversary in control of a previously compromised ECU can reprogram *another* ECU on the same CAN bus. Following the timeline depicted in Figure 22, we make the same assumption that our adversary can control the software on this compromised ECU and sniff bus traffic. The adversary then keeps the compromised ECU dormant until the victim vehicle goes to a dealership for a firmware update on the safety-critical target ECU. With the increase in recalls requiring firmware updates for newer vehicles [90], it is likely that ECUs will receive an update during the vehicle's lifetime. Here, the adversary uses the launchpad ECU to snoop and record the technician's communication with the target ECU as the technician performs a firmware update. After capturing the *valid* challenge-response pair, the adversary could perform a replay attack by forcing the target ECU to reproduce the same challenge and then sending the observed response. Thus, this adversary can pass authentication without requiring a brute-force attack or knowledge of the secret key.

The key here is forcing the target ECU to reproduce a previously observed challenge. Our main insight is to exploit a disconnect that enables the adversary to influence and control an ECU's source of randomness. Modern ECUs use processor uptime, or the runtime since the last reboot, as a randomness source for generating the challenge. Relying on time as a source of randomness is known to be a poor choice for seeding a PRNG [91, 92, 93]. However, we find that modern ECUs still use processor uptime and, thus, opens the door to exploit if an adversary controls when the uptime is reset. To grant an adversary control over the processor uptime, we develop a technique that exploits the ECUReset UDS service, an unprivileged (i.e., no prior SecurityAccess needed) command that requests an ECU to reset itself. Here, an adversary forces a target ECU to reproduce the same challenge by: (1) requesting the target to reset itself via UDS ECUReset, (2) waiting a specific amount of time linked to the observed challenge, and (3) requesting the challenge from the target ECU. While this technique alone can increase the likelihood of observing a specific challenge well beyond the expected likelihood, we repurpose our work on *CANnon* to develop a novel technique that grants an adversary finer timing control of their message transmissions, enabling a further increase in likelihood.

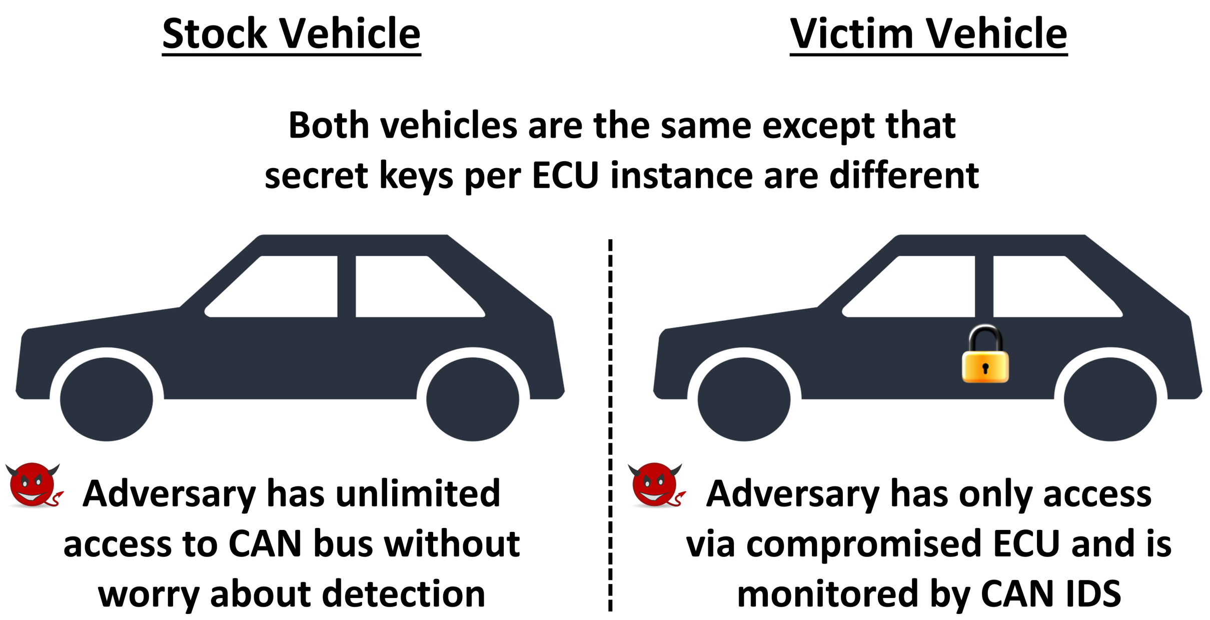

Stock Vehicle
Victim Vehicle
Both vehicles are the same except that secret keys per ECU instance are different
Adversary has unlimited access to CAN bus without worry about detection
Adversary has only access via compromised ECU and is monitored by CAN IDS

Figure 23: Access to stock vehicle vs. constraints on victim vehicle

Contributions: In summary, we contribute the following:

- We introduce new methods to control the weak source of randomness used by modern ECUs and launch replay attacks with a high rate of success. In contrast to previous work, we do not require any knowledge of the key or encryption algorithm.
- We demonstrate the ability to force a target ECU to reproduce a previously observed challenged. We also construct a novel method to increase the likelihood of repeating this same challenge.
- We bypass the authentication on a real powertrain ECU in less than a handful of attempts and demonstrate the ability to upload code to it. We also show how other powertrain ECUs and a gateway ECU are vulnerable to our technique by demonstrating a high likelihood of repetition for a given challenge on these ECUs.

5.3 Threat model and attack insight

To better understand the threat model we encounter here, we formally define the roles of the adversary and the victim. We first consider an adversary that wishes to perform an attack on a victim vehicle V_{victim} that is of model M. We also assume that the adversary (or any other party) can obtain a stock version of this model, which we can call V_{stock}. For all ECUs within V_{victim} and V_{stock} as depicted in Figure 23, all software and hardware components are the same except for secret keys; for example, the engine ECU in V_{victim} has a different secret key than the engine ECU in V_{stock}. Now consider that the adversary wishes to target a specific ECU, E_{target}, in V_{victim}, but E_{target} does not have any wireless interfaces. We now formulate the capabilities of all parties involved as the adversary attempts to attack E_{target}.

Adversary on compromised ECU: As E_{target} does not have any wireless interfaces and thus the adversary cannot remotely exploit it, the adversary must consider an alternative path to attack E_{target}. The adversary can instead remotely compromise another ECU in V_{victim} that has some wireless interface. This ECU, $E_{compromised}$, can serve as a "launchpad" for the adversary to use to ultimately target E_{target}. As a result of the shared design of vehicles across the same model M, the adversary can learn from and practice on their V_{stock} prior to launching the attack on V_{victim}. Here, we assume that the adversary can successfully perform a remote compromise of the software on $E_{compromised}$ via some exploit. We also assume that the adversary can stealthily install any attack code onto $E_{compromised}$ without detection. Additionally, it is impractical for the adversary to have any physical access to V_{victim}, especially if the adversary wishes to launch an attack at scale, so the adversary can only control the application layer of $E_{compromised}$. Also, $E_{compromised}$ does not need to run a UDS application, but it should be capable of sending CAN messages with UDS-related content. The adversary can also use $E_{compromised}$ to record CAN traffic at any time after the initial remote compromise succeeds.

Target ECU in victim vehicle: Since the adversary likely wants to achieve a high-impact attack, we assume they will choose an E_{target} that can control safety-critical functions in V_{victim}.

We assume that E_{target} runs a standard implementation of UDS, includes the SecurityAccess service to enable another device to achieve authentication on E_{target}, and implements the RequestUpload service to enable a technician (and an adversary if authentication is bypassed) to reprogram the ECU. Furthermore, we assume that the adversary cannot directly attack this ECU as it has no wireless interfaces and that any attempt to send critical diagnostic commands to E_{target} do not function over a low vehicle operation speed threshold. We also assume that E_{target} will require a firmware update at some point during the life of V_{victim}, which will allow the adversary to capture a valid authentication.

Firmware update by technician: Following the increasing number of firmware-related recalls that require a firmware remedy [90], we assume that V_{victim} will be taken to a technician at the dealership at least once during the vehicle's lifetime. The technician will use a specific tester tool for the model M of V_{victim} and can communicate with all UDS-enabled ECUs in the vehicle, including E_{target}. We assume that the technician follows a typical workflow of connecting to the vehicle and updating the firmware for E_{target} as part of the recall or repair. The technician's tool will communicate with E_{target} over the CAN bus on V_{victim} and initiate the UDS SecurityAccess service. Here, the tool will request a challenge from E_{target}, which responds with the requested challenge. The tool then uses this challenge and the secret key to compute a response, which it sends to E_{target}.

IDS on victim vehicle's CAN bus: We assume that the CAN bus on V_{victim}, B_{victim}, is unmodified from the factory and that the traffic over this bus has similar characteristics (e.g., message periodicity, bus load) to the CAN bus of V_{stock}, B_{stock}. We also assume that all ECUs in V_{victim} have full read-write access to the bus, including $E_{compromised}$ and E_{target}. To protect this network against an adversary, we assume that B_{victim} is monitored by a software-based IDS (as detailed in Chapter 2) that monitors the number of failed attempts [94] to authenticate via SecurityAccess. We assume that the IDS will alert to an attack based on a specific threshold that is dependent on the model M of V_{victim}. As UDS SecurityAccess authentications rely on attaching a new device (i.e., the tester) to the CAN bus, it is likely that an IDS will be configured to detect a malicious tester by tracking its CAN bus transmissions. We assume that it will instead rely on the built-in maximum number of attempts as a method of detecting an unauthorized attempt to authenticate [94].

5.3.1 High-level insight

The ultimate goal of our adversary is to successful authenticate via the SecurityAccess service running on the target ECU. To achieve this, our adversary must provide the correct response to a given challenge. In addition to achieving this goal, our adversary likely wants a method that is generalizable to arbitrary ECUs (i.e., our method should not be unique to any given model, where prior work was model-specific [52]) and one that is undetectable by current IDS techniques (i.e., our method should not require brute force or alter non-diagnostic messages). Considering these limitations, one type of attack that could succeed here is a replay attack for two major reasons: (1) a replay attack does not rely on any model-specific security algorithms, especially if the adversary could replay a previously-observed response, and (2) a replay attack that can succeed within just a few attempts would easily bypass an IDS that monitors for a handful of failed authentication attempts. The main challenge we face with a replay attack is that the random challenge makes replay difficult; we can only do a replay attack if a victim ECU produces the same challenge.

It is clear that, if the random challenge were no longer random but fixed, the adversary would have a viable attack that could achieve their goals. To identify how ECUs construct their source of randomness, we should look to other domains that implement challenge-response authentication [95]. In general, randomness sources typically utilize either processor runtime [93, 96], keyboard/mouse input [97], and static random-access memory (SRAM) start-up noise [62]. While some ECUs could make use of the true random number generators (TRNGs) that would be resident on an ECU's Trusted Platform Module (TPM) or Hardware Security Module (HSM), the cost of this added hardware can be expensive and thus is typically reserved for gateway and infotainment ECUs rather than safety-critical ECUs (e.g., powertrain ECUs). Out of this selection of potential randomness sources, we decided to investigate the most straightforward approach:

processor runtime.

Adversary can request a hard reset: To force an ECU to produce a fixed challenge, we must find some attack technique that can force an ECU's source of randomness to produce the same random value, or nonce, each time. With regard to processor runtime, we consider two possibilities in how this randomness source is implemented: (1) processor runtime is persistent and keeps track of either real wall clock time or total time that the ECU has run even when power cycled (i.e., powered-off and then powered-on), or (2) processor runtime is reset every time the ECU is powered cycled and actually represents *processor uptime*. If the latter option were true, then performing a hard reset (i.e., one that power cycles the ECU) would have some influence over the source of randomness. By exploring the available UDS services and options under each command, we find that a hard reset option exists for the UDS ECUReset service, and it does not require the adversary to pass authentication before requesting a hard reset.

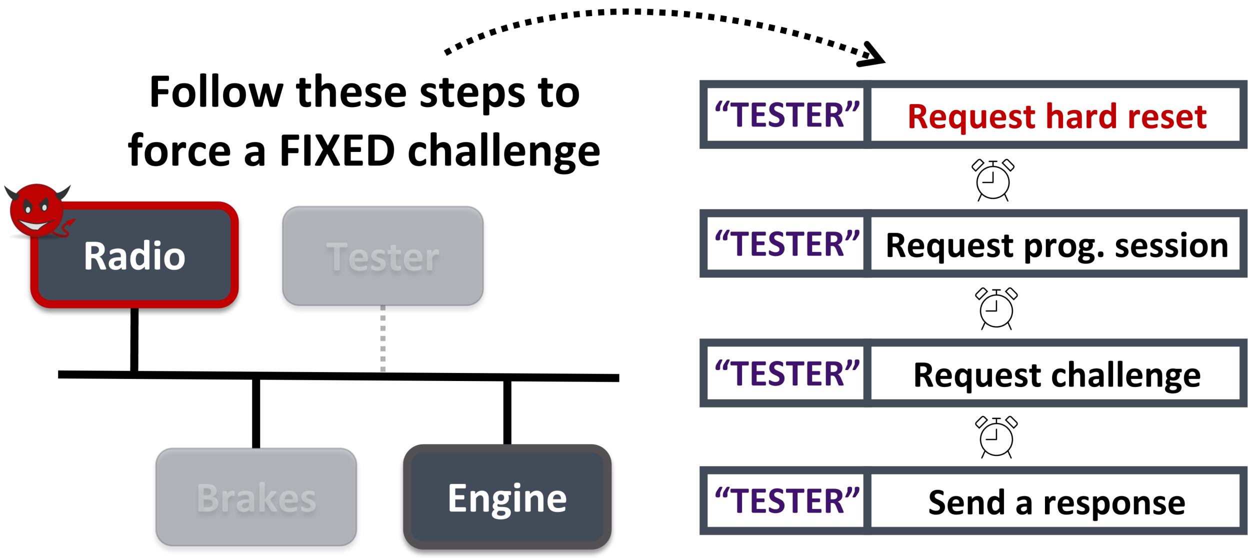


Figure 24: Steps to exploit the hard reset request

To ensure that our ECUs use processor uptime as their source of randomness, we can build a simple test to explore the potential for exploiting the hard reset command. We can mimic the diagnostic tester by using the tester's message ID and sending the following request as depicted in Figure 24: (1) request a hard reset of the target ECU, (2) request that ECU to enter into a programming session, and then (3) request a challenge from the target ECU. Instead of sending a response, we simply track what challenges we observe from the target ECU and perform this experiment 1000 times. For one of our ECUs that uses a 3-byte challenge, we should expect a challenge to be reproduced with a likelihood of 1 out of 2^{24}; our experiment resulted in some challenge appearing at over 5% of the total number of captures. From this basic test, it is a clear indication that we identify a vulnerability, where the hard reset command can be exploited to influence an ECU's source of randomness when producing a challenge. Finally, to confirm this insight that ultimately enables the *CANdid* attack, we analyzed actual implementations of the SecurityAccess service from real ECUs and confirmed our findings with access to real source code. It then became evident that modern ECUs across manufacturers use processor uptime as their source of randomness.

Requirements to proceed with attack: Following this attack insight, we clarify a few requirements that must be in place for this attack to succeed. First, we assume that the involved ECUs (both compromised ECU and victim ECU) implement a standard UDS implementation, which we find to typically include, at a minimum, the following services: UDS DiagnosticSessionControl, UDS ECUReset, UDS SecurityAccess, UDS RequestUpload, and UDS TransferData. The victim ECU's UDS implementation must also offer the hard reset option for ECUReset, and the victim ECU must use processor uptime as its source of randomness, which we find true across two different OEMs. Additionally, the ECUReset service be accessible to any other ECU as the remote adversary will compromise an in-vehicle ECU to perform the authentication bypass attack. Another requirement is that the adversary should have knowledge of the UDS server and UDS client IDs for the victim and tester, respectively. The adversary can identify all UDS-capable devices on a CAN bus by simply request a DiagnosticSessionControl service with all possible client IDs between 0x700 and 0x7FF for standard CAN implementations; a positive response to this service request will indicate an available UDS server, which will typically be the UDS client ID plus 0x040.

5.3.2 Single known-plaintext attack

Considering the threat model and our attack insight, we now introduce our plan of attack to perform an authentication bypass. Since the remote adversary has the ability to simply lay dormant and sniff the bus using the initial compromised ECU, we envision that our adversary simply waits until an authorized technician's diagnostic tester is connected. While the tester authenticates with the victim ECU and proceeds to perform a code upload to it, the adversary can capture just a single valid challenge-response pair. Typically, if the challenge were random and not controlled by the adversary, it would be impractical for the adversary to utilize this single challenge-response pair. A random challenge prevents an adversary from simply requesting a challenge and then providing the same observed response; the victim ECU will expect a different response as the security algorithm to produce the response takes the random challenge as input.

However, if our adversary could exploit this hard reset insight to control the challenge that the victim ECU produces, then the adversary could potentially launch a replay attack. Here, the two main steps are: (1) the adversary must force the previously-observed challenge from the single captured pair to appear from the stock copy of the victim ECU and (2) then the adversary must design a technique to greatly increase the likelihood for the victim ECU to produce a given challenge. This latter stage will be critical when launching the attack on the victim vehicle as an IDS will be present and will try to detect numerous failed authentication attempts. As an example, if the adversary can eventually force an ECU to produce the previously-observed challenge with a 20% likelihood, then the adversary can authenticate with just five attempts (on average) and deceive a monitoring IDS.

5.4 Controlling the challenge

We now explore how a remote adversary can exploit this vulnerability to force an ECU to produce a previously-observed challenge with some likelihood. Here, we aim to observe a particular challenge with a higher likelihood than other challenges.

As we demonstrated in our attack insight, performing a hard reset before requesting a challenge can cause some challenge to occur at a higher-than-expected likelihood. To understand how the adversary can control the most-repeated challenge, we must first understand how the challenge is created. Here, the value of the challenge produced by an ECU is an input of the nonce from the randomness source and potentially some other fixed value (depending on the algorithm used, such as a linear congruential generator (LCG)). As a result, every change in the nonce will produce a change in the produced challenge. Thus, we can expect that every increase in the processor uptime will change the challenge that we receive from a victim ECU.

With this in mind, we can now consider the first major step for the adversary. Given the single challenge-response pair observed during the technician's authentication with the victim ECU, the adversary must first determine the necessary attack setup and parameters to force the victim ECU to produce the same observed challenge at least once. Then, the adversary can focus on what steps are necessary to increase the likelihood of this particular challenge and make it the most likely to appear. A simple experiment to run as we progress through each part of our approach follows the same one we did above: simply request 1000 challenges after a hard reset request and calculate which and how often a particular challenge appears from the victim ECU.

5.4.1 Selecting a challenge

It is clear that the challenge produced by an ECU is a function of the processor uptime. However, one obstacle that the adversary encounters here is that it is not sufficient to simply request a hard reset and then immediately request a challenge. As detailed above and in Chapter 2, there are other necessary steps that must occur between the reset request and the challenge request, namely the request to enter the programming session. On top of this, each request (whether a hard reset or diagnostic state change) must be acknowledged over the CAN bus, which is then followed by the completion of that request. For example, the timing of a reset request follows Figure 25. Here, we see that the reset request sent by the target is followed by a response from the

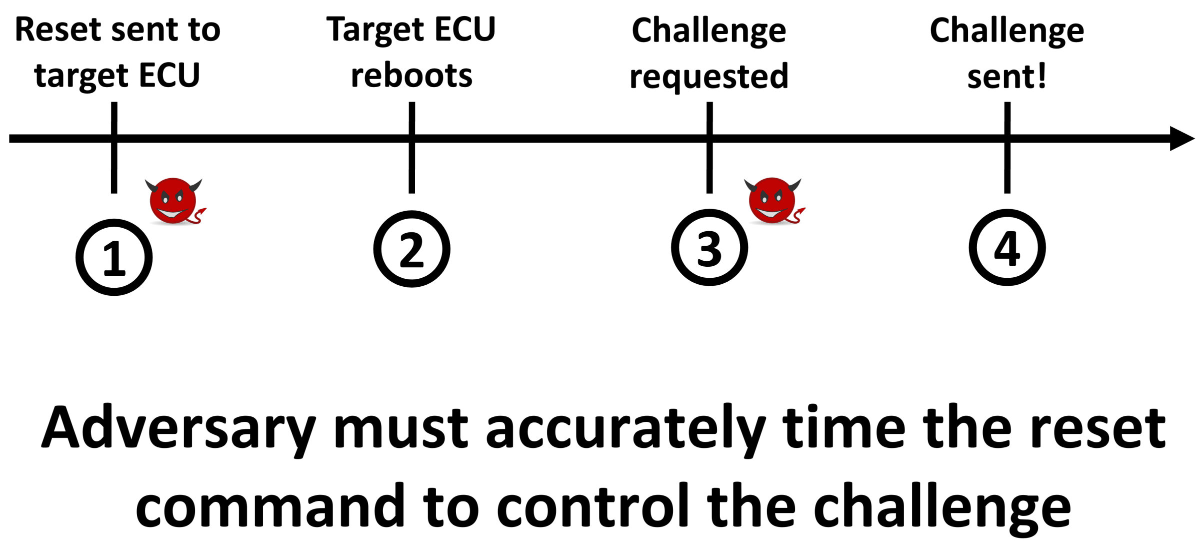


Figure 25: Overview of a single reset-controlled challenge request

target ECU acknowledging the request and then the actual reset of that ECU. The time between each of these steps can vary for each make and model of ECU, thus it is necessary to determine how to control the time between each step. For simplicity, we define several key timestamps during a potential attack:

- $t_{req(reset)}$, or the time when the adversary requests a reset.
- $t_{resp(reset)}$, or the time when the target acknowledges the reset request.
- t_{reset}, or the true time the target performs the reset.
- $t_{req(prog)}$, or the time when the adversary requests a programming session.
- $t_{resp(prog)}$, or the time when the target acknowledges the session change request.
- t_{prog}, or the time when the target enters the programming session.
- $t_{req(chal)}$, or the time when the adversary requests a challenge.
- $t_{resp(chal)}$, or the time when the target responds with a challenge.

Following this notation, each challenge produced by a target ECU will depend on the value of the time between the hard reset request and the challenge response, or $t_{req(chal)} - t_{req(reset)}$.

Ideally, $t_{req(chal)} - t_{req(reset)}$ should have as close to zero variance as possible; but, in reality, there are many factors that impact this value depending on the particular make and model of ECU:

- The value of $t_{reset} - t_{req(reset)}$ can have nonzero variance if the target ECU is actively transmitted other (likely periodic) CAN traffic. One approach here to reduce this variance is to request a programming session both before and after the hard reset request. While the session request after is expected, the session request before the reset likely reduces other software operations that occur on the target ECU. As a result, when the target ECU then receives a hard reset request, it can perform that operation with priority (since no other actions to take), thus producing a more consistent value for $t_{reset} - t_{req(reset)}$.
- The value of $t_{req(prog)} - t_{reset}$ has a minimum time depending on the target make and model ECU as an ECU needs sufficient time to actual perform the hard reset before it can accept any UDS service request. As the hard reset simulates a power cycling of the ECU, all of the software must reinitialize. As a result, more software to reinitialize after a hard reset on an ECU will cause $t_{req(prog)} - t_{reset}$ to increase. This finding is further supported by the fact that later model ECUs have a larger $t_{req(prog)} - t_{reset}$, which we suspect is due to added software complexities. We also see the same impact for the value of $t_{req(chal)} -$ as an ECU will need to run some software instructions to enter the programming session. While doing this, it will ignore any other UDS service requests. We also find here that $t_{req(chal)} -$ can be different between makes and models of ECUs.

After identifying the unique parameters for the hard reset and programming session state change requests that depend on the target ECU's make and model, we now attempt to exhibit some control of the exact challenge we observe from an ECU. As mentioned before, each value of $t_{req(chal)} - t_{req(reset)}$ will map to a different challenge based on the challenge algorithm. Unfortunately, the main hurdle we face here as depicted in Figure 26 is a result of the CAN bus timing resolution. In this example, the timing resolution of the CAN bus is 4μs while the resolution for

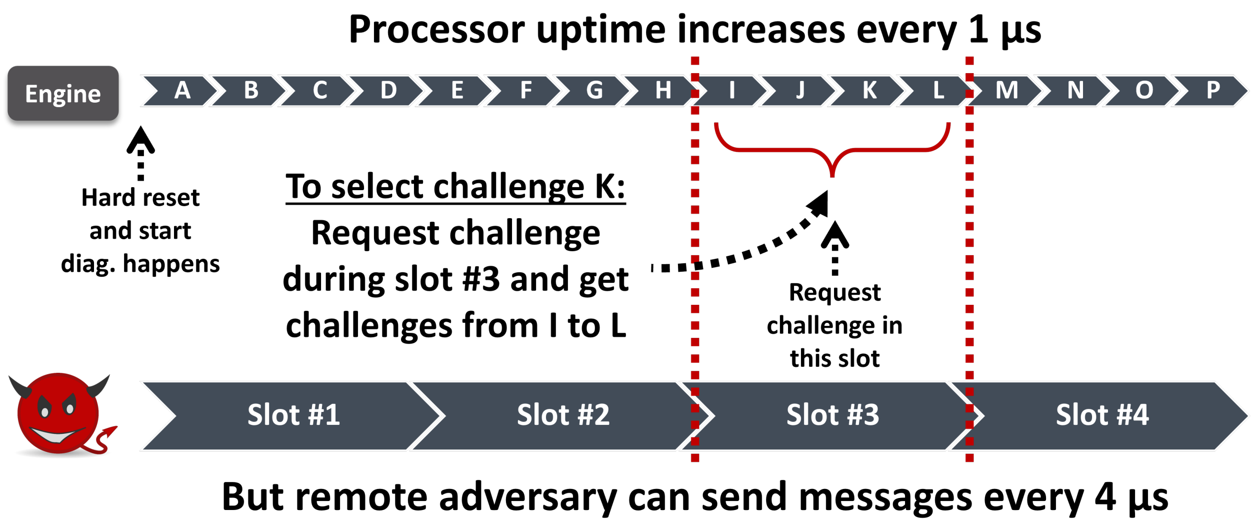

Processor uptime increases every 1 µs
Engine
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Hard reset and start diag. happens
To select challenge K:
Request challenge during slot #3 and get challenges from I to L
Request challenge in this slot
Slot #1
Slot #2
Slot #3
Slot #4
But remote adversary can send messages every 4 µs

Figure 26: Timing resolution of CAN bus versus victim ECU

processor uptime on the target ECU is $1\mu s$ (in reality, the processor uptime is much smaller). If our goal as the adversary is to get challenge K to appear on the bus, then we should send a challenge request during CAN bus slot time #3. However, even if we do this, we can get one of four possible challenges ranging from I to L, where I and L refer to clock cycles on the victim ECU as show in Figure 26. Likewise, if we send a challenge request during slot time #4, we should expect to get a challenge ranging from M to P. From this example, it is clear that we can select a particular challenge by simply waiting a sufficient time until a known CAN bus slot time will produce the expected challenge. However, determining how long to wait to send a challenge request is a difficult problem. To solve this, we must consider the variety of implementation of challenge algorithms as they each will have a slightly different approach to controlling the challenge.

5.4.2 Approach to different challenge algorithms

We now detail a variety of approaches that we can take to control the challenges, which will depend on the implemented challenge algorithm.

Linear and small space: For many makes and models of ECUs, it is common to find that the challenge algorithm is a linear-feedback shift register (LFSR) as seen in prior work [52]. If the challenge space (i.e., the number of bits in the challenge) is small enough, we can "reverse-engineer" the LFSR using the Berlekamp-Massey algorithm [98]. Here, the adversary can essentially build a rainbow table where, for each value of $t_{req(chal)} - t_{req(reset)}$, the adversary can know which challenge will be produced. Using the known LFSR, the adversary can determine the entire expected sequence and thus know how the challenge changes given an increment in the processor uptime value. We find that this challenge algorithm is common and appears on both powertrain ECUs that we experiment with.

Linear and large space: Suppose we had a similar design as above: an LFSR generating the challenge but the challenge space was significantly large enough that we could not reverse-engineer the LFSR algorithm itself. Our insight here is for the adversary to use our basic experimental setup (i.e., perform and track 1000 requests for a challenge) but continuously modify the value of $t_{req(chal)} - t_{req(reset)}$. We know that there will be one specific challenge that appears at a maximum likelihood for every time slot in our challenge request. If we increment $t_{req(chal)} - t_{req(reset)}$ by one time slot at a time, we get shift the likelihood to the next challenge that would be produced by this LFSR. Of course, if the processor uptime time slots are much smaller than the CAN bus bit width, then we may skip a few challenges as we increment $t_{req(chal)} - t_{req(reset)}$ by a single unit. While our approach for linear and small challenge space was sufficient for our powertrain ECUs, we implement this approach on those ECUs as well.

Non-linear and small space: As the challenge algorithm here is non-linear, we cannot expect the sequence of challenges produced as $t_{req(chal)} - t_{req(reset)}$ increases to be obvious. However, if the adversary could somehow predict approximately when an ECU would be reset, the adversary could reduce the potential values for $t_{req(chal)} - t_{req(reset)}$ to a reduced range of time, effectively reducing the search space for our adversary. One idea to predict this reset time is to analyze the process a diagnostic tester follows to upload code to an ECU. Using the OEM diagnostic tester we acquired, we noticed that the tester software requested us to manually power off and power on the ECU just prior to uploading new code. Here, the reset of the ECU will take place in a much smaller period of time (scale of a few seconds) compared to a vehicle that is sitting for a while and then having an ECU be updated (on the scale of tens of seconds). While we cannot reasonably map each challenge to its $t_{req(chal)} - t_{req(reset)}$ value, we can now build a table of challenges to $t_{req(chal)} - t_{req(reset)}$ values based on the expected time that a tester would normally request a manual hard reset.

Non-linear and large space: In addition to the challenge algorithm being non-linear, a large challenge space poses another hurdle for the adversary. If the remote adversary had a command-and-control setup, the adversary could measure the estimated $t_{req(chal)} - t_{req(reset)}$ from when the technician performed the single authentication. The adversary could measure this by timing when the ECU wakes up to when the tester's challenge request occurs. If the adversary transmits the estimated $t_{req(chal)} - t_{req(reset)}$ to an adversary-controlled server, the adversary can follow the steps for the above scenario and simply build a map of challenges to $t_{req(chal)} - t_{req(reset)}$ values but within the range of the estimated $t_{req(chal)} - t_{req(reset)}$. Another approach the adversary can consider here is actually controlling when the hard reset occurs before the tester requests a challenge. As the adversary is already sniffing on the bus, they could simply inject a hard reset request just before the authentication occurs. In our experiments, we find that the tester will often transmit some preliminary UDS commands prior to uploading code. If the adversary uses these commands as a trigger, they could simply insert a hard reset command just before the tester will request a programming session and then request a challenge.

5.5 Repeating a challenge

Now that the adversary can get the captured challenge to appear on the bus, the adversary will need to ensure that this challenge can appear with as high a likelihood as possible. To avoid detection by an IDS that alerts when a handful of failed authentication attempts occur, we must optimize our technique to requesting a specific captured challenge.

Potential for detection by an IDS: While the adversary can perform any amount of testing on their stock vehicle, the adversary is severely limited once performing attacks on the victim vehicle. We make the assumption that this vehicle is monitored by an IDS that will alert if the number of failed authentication attempts exceeds a set threshold. In practice, we find that this threshold is often, at a minimum, just five attempts. As a result, we can guarantee the success of an attack if the adversary can force an ECU to produce a previously-captured challenge to appear at over 20% likelihood. Also, compared to other work that required brute-forcing an ECU to extract secret keys [52], our technique can still succeed even if a vehicle implements diversified keys. By performing a capture-replay attack that follows a single known-plaintext attack model, we can limit the amount of time and the traffic output that our adversary requires to bypass authentication on the victim ECU. Thus, by aiming to get a likelihood of over 20%, we can achieve sufficient results to bypass authentication on modern UDS SecurityAccess implementations.

5.5.1 Maximizing likelihood of repetition

When performing our attack from the compromised ECU, it is critical for the adversary to ensure that the $t_{req(chal)} - t_{req(reset)}$ value is consistent across attempts. In our experiments, we find that launching the *CANdid* attack using a Linux device versus a microcontroller unit (MCU) can induce a lower likelihood of repetition. We believe that the other software functions that run between our wait time for the $t_{req(chal)} - t_{req(reset)}$ value causes the noise for this value. As a result, the adversary must ensure that their method of tracking time on the compromised ECU is consistent. The adversary can rely on hardware timers to ensure high precision in the timing of their requests to the victim ECU.

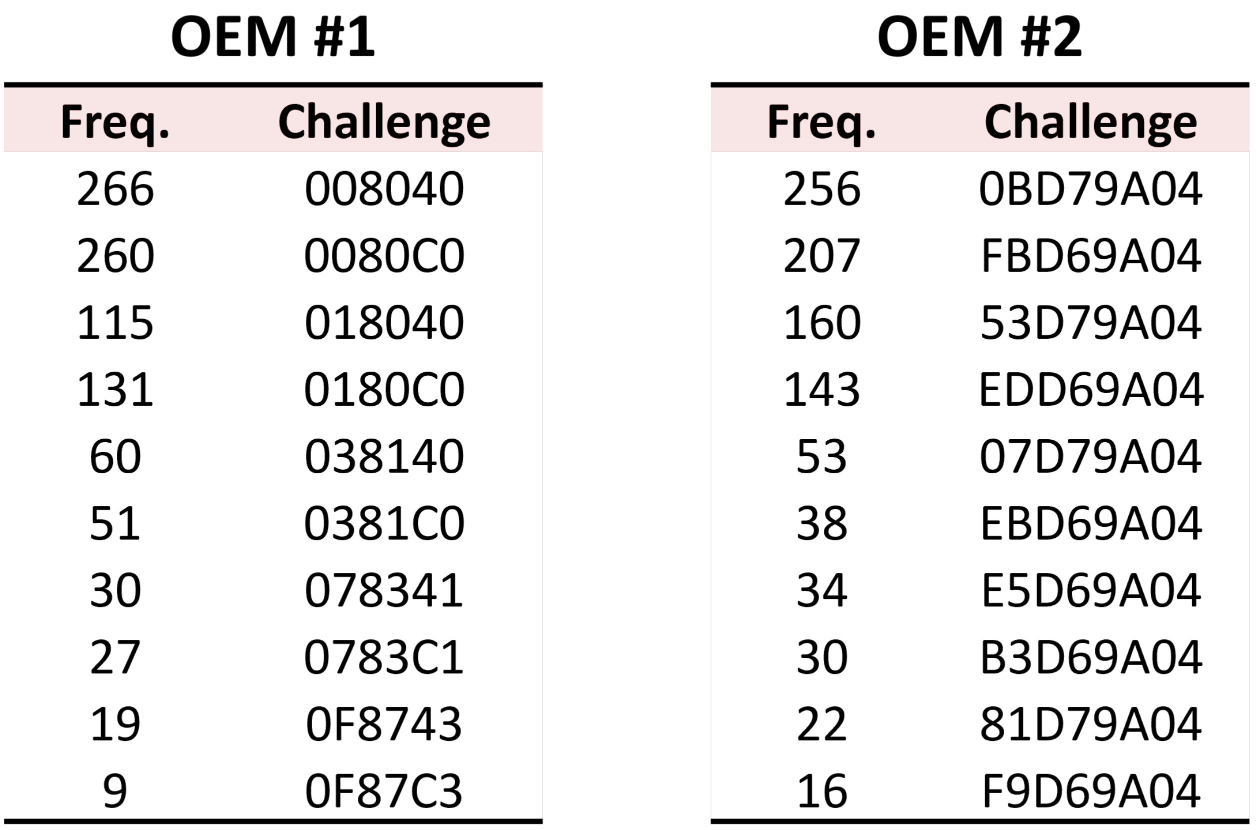
Timing control beyond a CAN bit width: The main limitation that the remote adversary faces in performing the *CANdid* attack is the timing resolution for sending requests on the CAN bus. As depicted in Figure 26, the CAN bus time slot can be many times larger than the time slots for the processor uptime. On a CAN bus operating at 500Kbps, the best timing resolution that the adversary can achieve is 2μs. In our experiments with an automotive-grade microcontroller, we find that the processor uptime could increment as fast as 11.9ns for an MCU with a clock speed of 84MHz (we note that processor uptime could increment at a multiple of the MCU's clock speed, but we assume worst-case scenario for now). This means that, for every challenge request over this CAN bus, the adversary could receive one out of *168* possible challenges. Thus, if the adversary were to perform this attack even with very consistent time between the hard reset request and the challenge request, the adversary would observe a specific challenge about 0.6% (or 1 over 168) of the time.

To get around this limitation, we identify a novel technique to reduce the timing granularity of our adversary when transmitting on the CAN bus. This technique makes use of the *CANnon* attack to impact the synchronization of the CAN bus to the adversary's advantage. For every CAN bus bit width, each bit is broken down into smaller segments called "quanta" at the physical-layer. These quanta are typically only necessary for the CAN controller to interpret the edges of a physical CAN signal and adjusting a controller's timing to the CAN bus. However, we find that using the *CANnon* to freeze for just a single quanta on the compromised ECU before it transmits a message can cause the entire bus to re-synchronize to this new transmission. Where the adversary previously could only control the timing of their messages down to 2μs for a 500Kbps bus, this technique can reduce that timing resolution down to 250ns (in this case, maximum number of quanta is 8, but this will change depending on the bus speed). With this new timing resolution, the adversary can now get finer control of when the challenge request is sent and thus increase the likelihood of forcing the victim ECU to produce a specific challenge.

5.6 Evaluation

In this section, we demonstrate the *CANdid* authentication bypass on a real powertrain ECU by: capturing a valid authentication, forcing the ECU to reproduce the previously-observed challenge, and then replaying the previously-observed response. We also demonstrate the ability to force an ECU to reproduce a challenge on another powertrain ECU and a gateway ECU.

Experimental setup: To demonstrate the significance of this attack, we launch *CANdid* against a variety of real ECUs from several real vehicles. In this work, we do not explicitly show the ability to compromise an in-vehicle ECU remotely as this has been the focus of a large number of prior works [10, 22, 35, 36, 37]. Rather, we build our attack on the assumption that these existing techniques would be successful in remotely compromising the software of an in-vehicle ECU. For our experiments, we physically connect to a CAN bus using an Arduino Due board to emulate a compromised ECU. In this work, we do not reveal the OEMs of each particular board as the disclosure process is ongoing. However, the ECUs that we use are used in several other works and are very common in the United States.



OEM #1

Freq.	Challenge
266	008040
260	0080C0
115	018040
131	0180C0
60	038140
51	0381C0
30	078341
27	0783C1
19	0F8743
9	0F87C3

OEM #2

Freq.	Challenge
256	0BD79A04
207	FBD69A04
160	53D79A04
143	EDD69A04
53	07D79A04
38	EBD69A04
34	E5D69A04
30	B3D69A04
22	81D79A04
16	F9D69A04

Figure 27: Challenges reproduced with over 25% likelihood

Authentication bypass on real powertrain ECU: We had access to a single diagnostic tester for just one OEM (we can refer to this OEM as OEM #1). With this tester, we identified that one of our powertrain ECUs from OEM #1 had a software update. To emulate the attack that our adversary would perform when on a real victim vehicle, we use the Arduino Due to sniff the CAN bus while we attempt to update the software on this powertrain ECU. Using the diagnostic tester, we first update the software to the latest version and capture a single challenge-response pair when the tester authenticates with the ECU. Using just this single pair, we can perform our attack. However, to obtain a potential payload to reprogram the ECU with, we downgrade the software version of the ECU and capture only the new code that was uploaded to the ECU. We then update the ECU again back to its latest version, where we will now try to perform our authentication bypass attack and downgrade the ECU's software version *without* using the tester.

Using the *CANdid* attack against this powertrain ECU from OEM #1, we achieve over a 25% likelihood of repeating any previously-observed challenge as depicted in Figure 27. The figure shows the results of our basic experiment (again, requesting and tracking 1000 challenges); with a likelihood of over 25%, we can expect to bypass authentication within four attempts, which is below the IDS detection threshold of five failed attempts. With this ECU, we perform a replay attack by forcing the ECU to produce the same challenge we observed during the tester's authentication. Once we see the same challenge appear from this ECU, we simply replay the associated response. We successfully received an "access granted" response from the ECU and gained access to privileged commands. Now the adversary can utilize the UDS RequestUpload service, which requires passing authentication, and can then use UDS TransferData to upload any code to the ECU. In this work, we simply upload the code for the older software version as a proof-of-concept. After performing this attack, we connect our diagnostic tester and observe that the current ECU version is the older version instead of the new version, proving that the *CANdid* worked.

Challenge repetition on other ECUs: As depicted on Figure 27, we saw a high likelihood of challenge repetition on another OEM, which we call OEM #2. This ECU is another powertrain ECU, and we perform the attack on both a 2016 and 2017 model year ECU. Where the challenges for this OEM are 4 bytes, we should expect the challenges to repeat with a likelihood of only 1 out of 2^{32} chance; however, we see an over 25% likelihood of repeating the same challenge. It is clear here that our *CANdid* techniques can achieve a sufficiently high challenge repetition on real ECUs. We also got access to an engineering copy of a real gateway ECU for another OEM. While we had limited access to this ECU, we were quickly able to demonstrate the *CANdid* attack on this ECU and achieved a 12.5% likelihood of repeating a specific challenge. This ECU had a 16-byte key so the expected likelihood of repeating a challenge should be 1 out of 2^{128}, which is astronomically smaller than the likelihood we achieved with *CANdid*. As a result, it is clear that *CANdid* can enable a capture-replay attack considering a single known-plaintext attack on real ECUs.

5.7 Countermeasures

We now discuss countermeasures against the *CANdid* authentication bypass attack. We consider two classes of solutions: ones that harden the authentication protocol to prevent such an attack and ones that aim to detect the attack in the future.

Hardening UDS SecurityAccess: One simple approach to defending against this attack is to place the UDS ECUReset service *behind* authentication. As long as the ECUReset service does not need to be accessed by unauthenticated users, then this measure can prevent an adversary from influencing the ECU's source of randomness. Another approach to harden the UDS SecurityAccess implementation is to seed the challenge with both a nonce *and* the ECU's unique key. Assuming that key diversification is in use, a challenge based on the unique key along with a nonce ensures that the challenges produced by each ECU instance are different from each other, preventing an adversary from planning an attack on a stock vehicle and then launching their attack on a victim vehicle.

Additionally, if key diversification is employed, then ECUs should implement a form of event-driven key rotation [99]; for example, the keys for an ECU should be updated whenever the firmware is upgraded. If a secure back-end server keeps track of the key, then this solution can be viable if the tester acts as a simple pass-through as we detailed in Chapter 2. Moreover, another approach to harder a SecurityAccess implementation is to use a better source of randomness. As highlighted in other work that uses processor runtime [93, 96], this source of randomness can be a poor choice compared to other sources, such as SRAM start-up noise [62]. Here, an ECU could use

SRAM, voltage, or other potentially random sources to generate a source of randomness instead of requiring the use of an expensive TPM or HSM.

Detecting the attack: Since the UDS requests originate from an in-vehicle ECU instead of an attached diagnostic tester, pinpointing the provenance of these messages could indicate a potential attack. One approach could use prior work on secure transceiver concepts [20] to filter outgoing UDS request messages for all in-vehicle ECUs. By blocking all requests that originate from in-vehicle ECUs, we can now expect that only a diagnostic tester can initiate UDS requests to in-vehicle ECUs. Of course, this defense assumes that ECUs do not need to request UDS services with each other, but the filter can be adjusted to permit just a few permitted services. Another approach lies with an observation from prior work on voltage-based IDSes [16, 17]. This prior work focused on tracking the voltage profiles of existing in-vehicle ECUs; if we did the same, we could detect an attack by identifying UDS services requests that have a similar voltage profile. If we expect the diagnostic tester connect to the CAN bus, it will have its own voltage profile that will be distinct from the in-vehicle ECUs, thus we could tell when a tester is physically connected.

5.8 Summary

In this chapter, we presented *CANdid*, an authentication bypass that can perform a single known-plaintext attack on a safety-critical ECU. *CANdid* leverages the reset diagnostic command to impact an ECU's source of randomness even though this PRNG was designed to be random for the purpose of making brute-force and replay attacks challenging. By exploiting this disconnect, *CANdid* can capture a single authentication and force an ECU to reproduce a previously-observed challenge, enabling a replay attack. With this attack, an adversary can then gain the appropriate access to upload code to a given ECU. We also propose countermeasures that target this disconnect by requiring authentication prior to permitting the reset diagnostic command or by using a PRNG that would be unaffected by a reset.

6 Kill-Chain Proof-of-Concept

To conclude the thesis contributions, we now detail a proof-of-concept demonstration of our attack kill-chain. Our goal here is to demonstrate how a remote adversary (limited to only running software instructions from a single compromised ECU) can upload code to another safety-critical ECU that would not have any remote interfaces for an adversary to directly exploit. As the goal of this proof-of-concept is to upload code to an ECU following our kill-chain stages, we select a target ECU that allows us to have access to two versions of code: an older version and the latest version. Our attack idea (as a proof-of-concept) is to downgrade the target ECU's software version.

6.1 Experimental setup

Out of all of our experimental ECUs, we find that a powertrain ECU from OEM #1 has two software versions that we can upload using a diagnostic tester for OEM #1.[13] Thus, we acquire a vehicle that is the same make as OEM #1. We attach our Arduino Due to the vehicle's OBD-II port and simply read the vehicle's CAN traffic (just as a remote adversary would). We then identify the set of ECUs and their source messages and find that there is a total of nine ECUs, including the powertrain ECU. As we did not have permission to launch this attack on this vehicle instance, we emulate this vehicle's network by building a surrogate network consisting of our powertrain ECU from OEM #1 and a total of eight Arduino Dues programmed to emulate the captured traffic of the non-powertrain ECUs (including IDs, transmission rates, and overall bus load). We select one of these Arduino Dues as our compromised ECU so we launch our attack from this Due. While launching our attack, we also transmit the original CAN bus traffic for this particular Due and ensure that all traffic follows the strict IDS rules detailed in Chapter 4.

[13]Using this tester, we can capture the software for the older version of code over the CAN bus and store this as our planned attack code to upload. An adversary can alter this code to achieve their desired final attack payload, but we do not investigate building a payload in this thesis.

6.2 A step-by-step demonstration

Our demonstration consists of five major steps in the attack timeline as depicted below:

1. By running the *CANvas* network mapper from the "compromised" ECU, we *confirm* that the eight other ECUs on the network (including the powertrain ECU) match exactly the network we expected from the real vehicle instance of the same make and model.

2. We connect the OEM #1 diagnostic tester to this surrogate network and update the powertrain ECU with the latest software version. While we perform this update, our compromised ECU sniffs the bus and captures a single challenge-response pair. This tester requires the technician to reset the powertrain ECU prior to the upload just a few seconds later.

3. Then, after the tester disconnects from the network, we can prepare to launch the authentication bypass. Since we can expect that the captured authentication pair occurs within a few seconds of the reset, we can build a small lookup table containing a list of potential challenges and their associated $t_{req(chal)} - t_{req(reset)}$ values by running experiments on the stock vehicle. We then store this table on the compromised ECU and have it select the appropriate $t_{req(chal)} - t_{req(reset)}$ for the challenge it captured from the tester. Using the *CANnon* technique, the compromised ECU can achieve a finer timing granularity when launching the *CANdid* part of the kill-chain.

4. After using *CANnon* to precisely time the challenge request (following the appropriate $t_{req(chal)} - t_{req(reset)}$ value), we simply replay the capture response, concluding the *CANdid* authentication bypass. We receive a positive response from the powertrain ECU, which enables us to request an upload and then upload the "attack" code.

5. We send a request to the powertrain's UDS RequestUpload service and receive a positive response. We then upload the older software version using UDS TransferData and succeed in our attack. To confirm, we connect the diagnostic tester and request the software version. Here, we find the older version is now programmed on this powertrain ECU.

7 Reflections, Lessons Learned, and Future Work

We now conclude the thesis with a summary of our contributions, reflections on how this work impacts related fields, the lessons we learned on identifying these elusive vulnerabilities, and future work to investigate.

7.1 Summary of contributions

By identifying disconnects between design assumptions and actual implementations, we can construct an attack kill-chain that enables a remote adversary to reprogram another in-vehicle ECU and, thus, informs countermeasures in the defense of next-generation vehicles.

7.1.1 *CANvas* contributions

In this thesis, we present the *CANvas* network mapper, which permits a remote adversary to passively sniff CAN bus traffic and extract the set of ECUs and which message IDs they send. We identify clock offset as a unique source ECU identifier that an adversary can extract from timestamped CAN traffic. With respect to our attack kill-chain, an adversary can use *CANvas* to map a stock version of the victim vehicle and then compare that map to the network map of the actual victim vehicle. We successfully identify a disconnect between the broadcast nature of the CAN protocol, which naturally includes no source information, and the periodicity of CAN traffic on real vehicles, which we exploit to identify a message's source ECU. As a result, this disconnect informs us that message periodicity reveals information on which ECU sends a given message ID so we propose countermeasures to intentionally disrupt this periodicity or make it difficult for an adversary to track messages by ID altogether. To demonstrate the consequence of this disconnect, we map a real 2009 Toyota Prius that had an additional transmitting ECU added to its network. If our remote adversary mapped an unmodified 2009 Prius, they would find that this Prius' network was different and could make a decision to abort their kill-chain if the added ECU would impact their attack.

7.1.2 *CANnon* contributions

For the *CANnon* disruption attack, we demonstrate how a remote adversary could exploit a new power-saving feature to impact a compromised ECU's CAN transmissions at the physical layer. We construct a technique that exploits software controls for peripheral clock gating to "freeze" an ECU's CAN transmission mid-message. With respect to our attack kill-chain, an adversary can use *CANnon* to disrupt other CAN transmissions and even trick other ECUs into shutting down their CAN interfaces. We successfully identify a disconnect between the CAN protocol hardware, which should enforce protocol compliance, and the peripheral clock gating feature on real MCUs, which we exploit to bypass protocol compliance and then inject individual bits on the CAN bus. As a result, this disconnect informs us that peripheral clock gating can be exploited to impact the CAN physical layer so we propose countermeasures to detect malicious usage of this feature or removing this feature altogether for just the CAN protocol. To demonstrate the consequence of this disconnect, we target and shut down a powertrain ECU on a 2017 Ford Focus. With this capability, our adversary can also impact the transmission time of other CAN messages and even the synchronization of all ECUs to the CAN bus.

7.1.3 *CANdid* contributions

Our *CANdid* authentication bypass demonstrates how a remote adversary can exploit a particular diagnostic command to enable a replay attack against the UDS SecurityAccess service. We construct an attack that exploits the UDS ECUReset service to request a hard reset that forces an ECU to reproduce a previously-observed challenge. With respect to our attack kill-chain, an adversary can use *CANdid* to bypass authentication on a safety-critical ECU and then permit the adversary to upload new software to this ECU. We successfully identify a disconnect between the processor uptime-based PRNG used in ECUs, which should provide sufficient randomness to prevent replay and brute-force attacks, and the hard reset command that requires no authentication, which we exploit to control the challenge produced by an ECU and then launch a replay attack. As a result, this disconnect informs us that the hard reset request can be exploited to bypass authentication via UDS SecurityAccess so we propose countermeasures to harden the authentication protocol (by enhancing the PRNG, requiring authentication to even request a hard reset, etc.) or detect when an in-vehicle ECU attempts to authenticate versus a diagnostic tester. To demonstrate the consequence of this disconnect, we capture a single valid authentication using an OEM tester and then perform a replay attack on a real powertrain ECU. With this capability, our adversary can complete our proposed attack kill-chain and use an initial compromised ECU to reprogram another safety-critical ECU on a given in-vehicle network.

7.2 Impact on related automotive fields

While we demonstrate this attack kill-chain in the field of automotive security, we highlight how this work can impact other related fields.

Heavy-vehicle (trucking) industry: While trucks differ from passenger vehicles in many ways, their in-vehicle networks can be similar. Heavy-vehicles also employ both the CAN and UDS protocols as well as similar hardware in their ECUs. As a result, it is likely that the trucking industry is vulnerable to our attack kill-chain. One major concern here is that the prospect of a remote adversary could be more likely against a truck. While passenger vehicles are typically left unmodified after production so a remote adversary must target an existing in-vehicle ECU, trucks often have an electronic logging device (ELD) attached directly to the CAN bus [100]. An ELD is typically sold by a separate company and could serve as an entry point to a vehicle's CAN bus. Where the truck manufacturer can focus heavily on security for their in-vehicle ECUs, an added ECU from a company that may not share the same security concerns could open the door to a remote attack on trucks. Then, by following our attack kill-chain, the impact of a remote attack could significantly increase.

Autonomous vehicle (AV) industry: AVs move the bar in technology by shifting to drive-by-wire systems, where we rely on electronics to control many (if not all) of a vehicle's safety-critical functions. As a result, we lose the mechanical "backups" that non-AVs had. As a result, even if a remote adversary could disable the brakes on a non-AV, the driver could still use their emergency brake to stop the vehicle (although not ideal); for an AV, however, such an attack would not give the driver any option as the safety-critical brakes are electronically controlled. While the AV industry will also introduce newer in-vehicle network protocols (e.g., Automotive Ethernet) for handling sensor data, it is likely that safety-critical systems will still operate over CAN so this scenario is still feasible for AVs. In addition to this, the AV industry reintroduces the notion of a physical attack. Considering the adoption of AV fleets, we will likely see situations where a malicious passenger will attach a device directly to the AV's CAN bus. Then, when this passenger leaves and another victim passenger enters the AV, the adversary can launch their attack. As a result, we need to investigate techniques to identify potentially malicious added components to a vehicle's CAN bus.

Electric vehicle (EV) industry: In their landmark work, Miller et al. identify that a vehicle worm would be an interesting and scary attack that could potentially cause significant damage [10]. While they focus on a vehicle worm that remotely compromises one vehicle and then scans for others and compromises those, we envision a similar attack but against the EV industry. Considering that EV chargers can communicate over the CAN bus [101], this access opens the door to a worm-like attack. Suppose an adversary could plant a worm on an EV charger that will attack a connected EV's CAN bus. Here, we envision a worm that spreads to EVs that then spread the worm to other EV chargers and so on. Considering the physical connectivity between the EV charger and the EV itself, an adversary could even plan a physical man-in-the-middle device to initiate the worm and gain access to the first vehicle's ECUs.

7.3 Lessons learned

We now discuss lessons learned from other fields of research and lessons we learned about identifying elusive vulnerabilities similar to those found in this thesis.

Lessons from other fields: We find that it is important to learn lessons from other fields to better secure in-vehicle networks. Prior work on encrypting passwords in a similar manner to challenge-response authentication for traditional security finds that the use of time is a poor choice for a source of randomness [93, 96]. While the key exploit that we encounter for the *CANdid* attack was that the adversary had access to a hard reset command, this attack would not have happened in the first place if a better source of randomness were in place. It is well-known that the field of automotive security is behind traditional security [7, 51], and we should focus on *not* repeating the poor choices made in other fields.

A better security testbed: In our thesis, we often use the Arduino Due microcontroller as our initial hardware for our experiments when exploring new attacks. We have often found that the use of such an automotive-grade MCU has enabled us to find these attacks; we now provide an example for each project. For *CANvas*, we initially used a CAN dongle that communicates via Linux's `socketcan` module, which produces software timestamps. In our original experiments, we were unable to identify the fact that periodicity revealed unique timing characteristics. However, when we ported our software to the Arduino, we were able to achieve a finer timing granularity. By mimicking hardware that would actually exist in a real vehicle instead using an "easy-to-use" CAN dongle via Linux, we were able to observe the minute differences in clock offset and enable our results (we also had similar experiences when performing our *CANdid* authentication bypass attack). With respect to *CANnon*, we found that our use of an automotive-grade MCU that implemented a CAN peripheral enabled us to find this vulnerability where prior work did not. While many CAN projects use a simpler version of the Arduino, we used the Due, which closely resembles a high-performance automotive MCU family. Here, we were able to find the software instruction for clock gating and identify its impact to enable the *CANnon* attack. Thus, it is clear that a poor choice in testbed could be a simple barrier to finding these elusive vulnerabilities.

7.4 Future work

This thesis focuses on building a kill-chain path from a compromised infotainment (or telematics) ECU to a safety-critical ECU. We do not address the initial compromise and the final attack code upload, thus future work should encompass these other kill-chain stages.

7.4.1 Exploiting remote interfaces

One major challenge that we do not address in this thesis is methods to remotely exploit the initial compromised ECU inside a vehicle. We have seen multiple examples of such remote exploits [10, 22, 23, 24]. Following the vehicle-agnostic goal of this thesis, it would be worth investigating general approaches to gaining remote control of an in-vehicle ECU. One approach could be to explore mobile application vulnerabilities [35, 36, 37] and see if automakers share any potentially vulnerable software between their implementations. Likewise, as automakers build back-end systems that connect to the telematics ECUs in their vehicles, we could explore vulnerabilities that are shared among this back-end software. Following our insight that UDS implementations are somewhat similar across makes and models, it would not be surprising to see the same commonalities across automaker back-end services, especially if those services are not built in-house. Another remote interface exploit that would be worth exploring are added devices to the CAN bus, which can include anything from ELDs for trucks, personal-use diagnostic testers, and insurance dongles. Here, an adversary could exploit vulnerabilities on these devices to ultimately gain access to a victim vehicle.

One recent approach against remote attacks comes from when Tesla responded to a remote attack on their vehicles [22]. In their response, Tesla pushed an update that implemented code signing on the gateway ECU, which would prevent an adversary from modifying maliciously modifying the code on this ECU. However, one type of attack that is still possible with signed code is a a return-oriented programming (ROP) chain attack via exploiting a buffer overflow. Here, an adversary could still perform malicious software instructions but could only call existing commands in the ECU's code. It would be worth investigating how a ROP-chain attack could potentially achieve our attack kill-chain and produce the same results.

7.4.2 Bypassing firmware defenses

In this thesis, we simply upload an older version of code to a victim ECU as our proof-of-concept attack. However, it would be critical to investigate how an adversary could upload malicious code in the presence of firmware defenses on the victim ECU. One obstacle (not a defense per se) is the use of error-detecting codes (typically, a cyclic redundancy check (CRC)) that ensure new uploaded code was correctly transmitted over the CAN bus. Without knowledge of this CRC, an adversary would not be able to modify the code and simply push it to the victim ECU. However, as prior work has reverse-engineered the code on ECUs [52], an adversary can find the CRC algorithm used by the victim ECU and use it to calculate the appropriate CRC. Then, the adversary can modify the code at-will and upload it to the victim with the correct CRC.

There is also another potential defense that could prevent an adversary from uploading code to a victim ECU: secure boot with signed code. Secure boot ensures the integrity of the firmware and software by checking that the code's signature matches the expected signature. However, recent work discussed how researchers could bypass the secure boot checks on an Android platform [102]. As a result, an adversary could use a similar technique against a victim ECU and bypass the secure boot check once malicious code is uploaded. Additionally, if the code is signed and this secure boot bypass is not a viable option, our adversary could still achieve authentication with the ECU and write to data identifier (DID) fields using UDS commands. While these fields typically set configurations for an ECU, an adversary could potentially find a buffer overflow exploit and enable a ROP-chain attack. One important note to consider is that secure boot may use public-key infrastructure (PKI), which is costly to implement on the scale of the automotive industry. While we are likely to see PKI implemented on telematics and infotainment ECUs (the ones with wireless interfaces), it is probable that safety-critical ECUs without external interfaces will not use PKI. As a result, a security mechanism that uses PKI would not be in place for these ECUs, thus our attack kill-chain could still be possible. The main challenge then is gaining the initial remote compromise.

Bibliography

[1] Jack Baylis, M Grayson, C Lau, G Gerstell, B Scott, and Jim Nicholson. Transportation sector resilience. National Infrastructure Advisory Council, 2015. Available at this link.

[2] Diana Furchtgott-Roth, Patricia S Hu, Long Nguyen, Sean Jahanmir, William H Moore, Demi Riley, Steve Beningo, Matthew Chambers, Sonya Smith-Pickel, Hoa Thai, et al. Pocket guide to transportation 2021. 2021. Available at this link.

[3] Sara Baldwin, Amanda Myers, Michael O'Boyle, and David Wooley. Accelerating clean, electrified transportation by 2035: Policy priorities. Policy, 2021. Available at this link.

[4] Stacy Davis and Robert Gary Boundy. Transportation energy data book: Edition 39. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2021. Available at this link.

[5] Kara Kockelman, Stephen Boyles, Peter Stone, Dan Fagnant, Rahul Patel, Michael W Levin, Guni Sharon, Michele Simoni, Michael Albert, Hagen Fritz, et al. An assessment of autonomous vehicles: traffic impacts and infrastructure needs. Technical report, University of Texas at Austin. Center for Transportation Research, 2017. Available at this link.

[6] John Martin and Arthur Carter. Nhtsa cybersecurity research. In 25th International Technical Conference on the Enhanced Safety of Vehicles (ESV) National Highway Traffic Safety Administration, 2017. Available at this link.

[7] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In Security and Privacy (SP), 2010 IEEE Symposium on, pages 447–462. IEEE, 2010. Available at this link.

[8] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In USENIX Security

Symposium, pages 77–92. San Francisco, 2011. Available at this link.

[9] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. black hat USA, 2014:94, 2014. Available at this link.

[10] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. Black Hat USA, 2015:91, 2015. Available at this link.

[11] LG Cripps. Electronics in cars. Electronics & Power, 16(11):394–398, 1970. Available at this link.

[12] Uwe Kiencke, Siegfried Dais, and Martin Litschel. Automotive serial controller area network. SAE transactions, pages 823–828, 1986. Available at this link.

[13] WP UNECE. Grva,"draft recommendation on cyber security of the task force on cyber security and over-the-air issues of unece wp. 29 grva.", 29. Available at this link.

[14] Iso/sae 21434:2021 road vehicles — cybersecurity engineering. Standard, International Organization for Standardization, August 2021. Available at this link.

[15] Kyong-Tak Cho and Kang G Shin. Fingerprinting electronic control units for vehicle intrusion detection. In USENIX Security Symposium, pages 911–927, 2016. Available at this link.

[16] Kyong-Tak Cho and Kang G Shin. Viden: Attacker identification on in-vehicle networks. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 1109–1123. ACM, 2017. Available at this link.

[17] Wonsuk Choi, Kyungho Joo, Hyo Jin Jo, Moon Chan Park, and Dong Hoon Lee. Voltageids: Low-level communication characteristics for automotive intrusion detection system. IEEE Transactions on Information Forensics and Security, 13(8):2114–2129, 2018. Available at this link.

[18] Wonsuk Choi, Hyo Jin Jo, Samuel Woo, Ji Young Chun, Jooyoung Park, and Dong Hoon Lee. Identifying ecus using inimitable characteristics of signals in controller area networks.

IEEE Transactions on Vehicular Technology, 67(6):4757–4770, 2018. Available at this link.

[19] Giampaolo Bella, Pietro Biondi, Gianpiero Costantino, and Ilaria Matteucci. Toucan: A protocol to secure controller area network. In Proceedings of the ACM Workshop on Automotive Cybersecurity, pages 3–8. ACM, 2019. Available at this link.

[20] Bernd Elend and Tony Adamson. Cyber security enhancing can transceivers. In Proceedings of the 16th International CAN Conference, 2017. Available at this link.

[21] Stefano Longari, Matteo Penco, Michele Carminati, and Stefano Zanero. Copycan: An error-handling protocol based intrusion detection system for controller area network. In ACM Workshop on Cyber-Physical Systems Security & Privacy (CPS-SPC'19), pages 1–12, 2019. Available at this link.

[22] Sen Nie, Ling Liu, and Yuefeng Du. Free-fall: hacking tesla from wireless to can bus. Briefing, Black Hat USA, pages 1–16, 2017. Available at this link.

[23] Zhiqiang Cai, Aohui Wang, Wenkai Zhang, M Gruffke, and H Schweppe. 0-days & mitigations: roadways to exploit and secure connected bmw cars. Black Hat USA, 2019:39, 2019. Available at this link.

[24] Minrui Yan, Jiahao Li, and Guy Harpak. Security research report on mercedes-benz cars. Black Hat USA, 2020:38, 2020. Available at this link.

[25] Michael Muckin and Scott C Fitch. A threat-driven approach to cyber security. Lockheed Martin Corporation, 2014. Available at this link.

[26] Tim Ring. Connected cars–the next target for hackers. Network Security, 2015(11):11–16, 2015. Available at this link.

[27] Gartner says by 2020, a quarter billion connected vehicles will enable new in-vehicle services and automated driving capabilities. Available at this link.

[28] The car in the age of connectivity: Enabling car to cloud connectivity. Available at this link.

[29] An assessment method for automotive intrusion detection system performance. Available

at this link.

[30] Kyong-Tak Cho and Kang G Shin. Error handling of in-vehicle networks makes them vulnerable. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1044–1055. ACM, 2016. Available at this link.

[31] 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. Available at this link.

[32] Sekar Kulandaivel, Shalabh Jain, Jorge Guajardo, and Vyas Sekar. Cannon: Reliable and stealthy remote shutdown attacks via unaltered automotive microcontrollers. In 2021 IEEE Symposium on Security and Privacy (SP), pages 195–210. IEEE, 2021. Available at this link.

[33] CAN Specification. Bosch. 1991. Available at this link.

[34] Obd-ii background information. Available at this link.

[35] Experimental security research of tesla autopilot. Available at this link.

[36] Car hacking research: Remote attack tesla motors. Available at this link.

[37] New car hacking research: 2017, remote attack tesla motors again. Available at this link.

[38] DJ Wise. Vehicle cybersecurity dot and industry have efforts under way, but dot needs to define its role in responding to a real-world attack. Gao Reports. US Government Accountability Office, 2016. Available at this link.

[39] Marco Di Natale, Haibo Zeng, Paolo Giusto, and Arkadeb Ghosal. Understanding and using the controller area network communication protocol: theory and practice. Springer Science & Business Media, 2012. Available at this link.

[40] Pal-Stefan Murvay and Bogdan Groza. Source identification using signal characteristics in controller area networks. IEEE Signal Processing Letters, 21(4):395–399, 2014. Available at this link.

[41] Chung-Wei Lin and Alberto Sangiovanni-Vincentelli. Cyber-security for the controller

area network (can) communication protocol. In 2012 International Conference on Cyber Security, pages 1–7. IEEE, 2012. Available at this link.

[42] Craig Smith. The Car Hacker's Handbook: A Guide for the Penetration Tester. No Starch Press, 2016. Available at this link.

[43] Sang Uk Sagong, Xuhang Ying, Andrew Clark, Linda Bushnell, and Radha Poovendran. Cloaking the clock: emulating clock skew in controller area networks. In Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, pages 32–42. IEEE Press, 2018. Available at this link.

[44] Mehmet Bozdal, Mohammad Samie, and Ian Jennions. A survey on can bus protocol: Attacks, challenges, and potential solutions. In 2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE), pages 201–205. IEEE, 2018. Available at this link.

[45] Tcan: Authentication without cryptography on a can bus based on nodes location on the bus. Available at this link.

[46] 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, page 71. ACM, 2017. Available at this link.

[47] 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. Springer, 2017. Available at this link.

[48] Hyun Min Song, Ha Rang Kim, and Huy Kang Kim. Intrusion detection system based on the analysis of time intervals of can messages for in-vehicle network. In 2016 international conference on information networking (ICOIN), pages 63–68. IEEE, 2016. Available at this link.

[49] Clinton Young, Habeeb Olufowobi, Gedare Bloom, and Joseph Zambreno. Automotive intrusion detection based on constant can message frequencies across vehicle driving modes. In Proceedings of the ACM Workshop on Automotive Cybersecurity, pages 9–14. ACM, 2019. Available at this link.

[50] Mahsa Foruhandeh, Yanmao Man, Ryan Gerdes, Ming Li, and Thidapat Chantem. Simple: Single-frame based physical layer identification for intrusion detection and prevention on in-vehicle networks. 2019. Available at this link.

[51] Charlie Miller and Chris Valasek. Adventures in automotive networks and control units. Def Con, 21:260–264, 2013. Available at this link.

[52] Jan Van den Herrewegen and Flavio D Garcia. Beneath the bonnet: A breakdown of diagnostic security. In European Symposium on Research in Computer Security, pages 305–324. Springer, 2018. Available at this link.

[53] Nastf vehicle security professional registry. Available at this link.

[54] Mert D Pesé, Troy Stacer, C Andrés Campos, Eric Newberry, Dongyao Chen, and Kang G Shin. Librecan: Automated can message translator. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 2283–2300, 2019. Available at this link.

[55] Charlie Miller. "interesting paper about mapping vehicle networks. we did it the old fashioned way, by physically disconnecting components and observing the effect. (or by putting a component into programming mode)", 2019. Available at this link.

[56] Konglin Zhu, Zhicheng Chen, Yuyang Peng, and Lin Zhang. Mobile edge assisted literal multi-dimensional anomaly detection of in-vehicle network using lstm. IEEE Transactions on Vehicular Technology, 68(5):4275–4284, 2019. Available at this link.

[57] Michele Russo, Maxime Labonne, Alexis Olivereau, and Mohammad Rmayti. Anomaly detection in vehicle-to-infrastructure communications. In 2018 IEEE 87th Vehicular

Technology Conference (VTC Spring), pages 1–6. IEEE, 2018. Available at this link.

[58] Tarun Yadav and Arvind Mallari Rao. Technical aspects of cyber kill chain. In International Symposium on Security in Computing and Communication, pages 438–452. Springer, 2015. Available at this link.

[59] Gordon Fyodor Lyon. Nmap network scanning: The official Nmap project guide to network discovery and security scanning. Insecure, 2009. Available at this link.

[60] Aleksandar Kuzmanovic and Edward W Knightly. Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants. In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pages 75–86, 2003. Available at this link.

[61] Pedro Garcia-Teodoro, Jesus Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. computers & security, 28(1-2):18–28, 2009. Available at this link.

[62] Daniel E Holcomb, Wayne P Burleson, and Kevin Fu. Power-up sram state as an identifying fingerprint and source of true random numbers. IEEE Transactions on Computers, 58(9):1198–1210, 2008. Available at this link.

[63] Junaid Haseeb, Masood Mansoori, and Ian Welch. A measurement study of iot-based attacks using iot kill chain. In 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pages 557–567. IEEE, 2020. Available at this link.

[64] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. Computer, 50(7):80–84, 2017. Available at this link.

[65] Fleet-ready electrification solutions. Available at this link.

[66] Green motors inc. Available at this link.

[67] Pre-programmed performance chips. Available at this link.

[68] Mohammad Farsi, Karl Ratcliff, and Manuel Barbosa. An overview of controller area network. Computing & Control Engineering Journal, 10(3):113–120, 1999. Available at this link.

[69] Ken Tindell, H Hanssmon, and Andy J Wellings. Analysing real-time communications: Controller area network (can). In RTSS, pages 259–263. Citeseer, 1994. Available at this link.

[70] Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. Real-Time Systems, 35(3):239–272, 2007. Available at this link.

[71] Introduction to can. Available at this link.

[72] Toyota techinfo service. Available at this link.

[73] Motorcraft info service. Available at this link.

[74] Jérôme Maye and Mario Krucker. Communication with a toyota prius. Available at this link.

[75] Tadayoshi Kohno, Andre Broido, and Kimberly C Claffy. Remote physical device fingerprinting. IEEE Transactions on Dependable and Secure Computing, 2(2):93–108, 2005. Available at this link.

[76] A. Van Herrewege, D. Singelee, and I. Verbauwhede. CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus. In ECRYPTWorkshop on Lightweight Cryptography 2011, 2011. Available at this link.

[77] Bogdan Groza, Pal-Stefan Murvay, Anthony Van Herrewege, and Ingrid Verbauwhede. Libra-can: A lightweight broadcast authentication protocol for controller area networks. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, Cryptology and Network Security, 11th International Conference, CANS 2012, volume 7712, pages 185–200. Springer, December 12-14, 2012. Available at this link.

[78] Microchip sam 3x family of mcus. Available at this link.

[79] Microchip sam v family of automotive mcus. Available at this link.

[80] Marcel Kneib, Oleg Schell, and Christopher Huth. Easi: Edge-based sender identification on resource-constrained platforms for automotive networks. Available at this link.

[81] Automotive semiconductor market - growth, trends, and forecast (2020 - 2025). Available at this link.

[82] Nxp mcus. Available at this link.

[83] Renesas mcus. Available at this link.

[84] Fujitsu mcus. Available at this link.

[85] Cypress mcus. Available at this link.

[86] Infineon mcus. Available at this link.

[87] Sam v71 xplained ultra evaluation kit. Available at this link.

[88] Spc58ec-disp discovery board. Available at this link.

[89] St spc5 family of automotive mcus. Available at this link.

[90] 2020 automotive defect recall report. Available at this link.

[91] Cwe-337: Predictable seed in pseudo-random number generator (prng). Available at this link.

[92] Cwe-338: Use of cryptographically weak pseudo-random number generator (prng). Available at this link.

[93] Encrypting passwords. Available at this link.

[94] Ramiro Pareja and Santiago Cordoba. Fault injection on automotive diagnostic protocols. escar USA, 2018. Available at this link.

[95] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. Handbook of applied cryptography. CRC press, 2018. Available at this link.

[96] Passphrase storage. Available at this link.

[97] Greg Taylor and George Cox. Digital randomness. IEEE spectrum, 48(9):32–58, 2011.

Available at this link.

[98] ER Berlekamp. Binary bch codes for correcting multiple errors. Algebraic Coding Theory, 1968. Available at this link.

[99] Key rotation. Available at this link.

[100] Alex Scott, Andrew Balthrop, and Jason W Miller. Unintended responses to it-enabled monitoring: The case of the electronic logging device mandate. Journal of Operations Management, 67(2):152–181, 2021. Available at this link.

[101] Gautham Ram Chandra Mouli, Johan Kaptein, Pavol Bauer, and Miro Zeman. Implementation of dynamic charging and v2g using chademo and ccs/combo dc charging standard. In 2016 IEEE Transportation Electrification Conference and Expo (ITEC), pages 1–6. IEEE, 2016. Available at this link.

[102] Breaking secure bootloaders. Available at this link.