CANflict: Exploiting Peripheral Conflicts for Data-Link Layer Attacks on Automotive Networks

Alvise de Faveri Tron
a.de.faveri.tron@vu.nl
Politecnico di Milano
Milan, Italy
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Stefano Longari
stefano.longari@polimi.it
Politecnico di Milano
Milan, Italy

Michele Carminati
michele.carminati@polimi.it
Politecnico di Milano
Milan, Italy

Mario Polino
mario.polino@polimi.it
Politecnico di Milano
Milan, Italy

Stefano Zanero
stefano.zanero@polimi.it
Politecnico di Milano
Milan, Italy

ABSTRACT

Current research in the automotive domain has proven the limitations of the Controller Area Network (CAN) protocol from a security standpoint. Application-layer attacks, which involve the creation of malicious packets, are deemed feasible from remote but can be easily detected by modern Intrusion Detection Systems (IDSs). On the other hand, more recent link-layer attacks are stealthier and possibly more disruptive but require physical access to the bus. In this paper, we present CANflict, a software-only approach that allows reliable manipulation of the CAN bus at the data link layer from an unmodified microcontroller, overcoming the limitations of state-of-the-art works. We demonstrate that it is possible to deploy stealthy CAN link-layer attacks from a remotely compromised ECU, targeting another ECU on the same CAN network. To do this, we exploit the presence of *pin conflicts* between microcontroller peripherals to craft *polyglot frames*, which allows an attacker to control the CAN traffic at the bit level and bypass the protocol's rules. We experimentally demonstrate the effectiveness of our approach on high-, mid-, and low-end microcontrollers, and we provide the ground for future research by releasing an extensible tool that can be used to implement our approach on different platforms and to build CAN countermeasures at the data link layer.

CCS CONCEPTS

• **Security and privacy** → **Hardware attacks and countermeasures**; • **Networks** → *Cyber-physical networks.*

KEYWORDS

Automotive Security; Controller Area Network; Hardware Attacks; Polyglot Frames; Conflicting Peripherals

ACM Reference Format:
Alvise de Faveri Tron, Stefano Longari, Michele Carminati, Mario Polino, and Stefano Zanero. 2022. CANflict: Exploiting Peripheral Conflicts for Data-Link Layer Attacks on Automotive Networks. In *Proceedings of Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3548606.3560618

1 INTRODUCTION

Nowadays, vehicles are equipped with an enormous amount of electronic devices [41], which can include WiFi access points, Bluetooth modules, cellular communication modules, gateways, telemetry systems, and dozens of Electronic Control Units (ECUs) [18]. A modern vehicle, even if not fully-featured, typically has well over 100 ECUs, with an estimated 7000 signals to transmit internally [22, 36]. To coordinate communication among ECUs, in-vehicle networks employ several kinds of bus protocols. The most prevalent and de-facto standard of such protocols is CAN. Developed in the 1980s, the CAN protocol was primarily designed for reliable and fast communications in noisy environments, without much consideration for security aspects. The lack of encryption, authentication, and integrity checking makes CAN bus networks vulnerable to different attacks. Such attacks have first been proven possible through on-board attack surfaces [8, 19, 26] and then demonstrated feasible from remote [17, 28, 35], and consist mainly in forging packets from exploited or malicious ECUs, which limits the capabilities of the attacker to those attacks that can be implemented through sending valid CAN frames on the bus. More recent attacks, however, such as the one described in [32], have demonstrated that vulnerabilities also exist at the CAN data link layer. These attacks are more powerful and harder to detect, but they are feasible only given that the attacker can inject bits that break the protocol rules at a low level (e.g., if it is possible to write on the bus while another ECU is writing at the same time). In practical scenarios, this requires high precision, especially on a high-speed CAN bus, which is hardly possible on a resource-constrained microcontroller, such as those found in ECUs. Hence, up to now, link-layer attacks against CAN were considered feasible only if the attacker had physical access to the CAN bus, with the exception of the work by Kulandaivel et al. [21], which, however, comes with significant limitations.

In this paper, we take a step forward by presenting CANflict, a novel approach to link-layer attacks against CAN that exploits the presence of peripherals connected to the same physical pins of the CAN controller (i.e., pin conflicts) to send and receive bits on the CAN bus on its behalf. These peripherals can be enabled and disabled by accessing dedicated memory-mapped registers from software, making this approach completely applicable without any hardware modification. Since conflicting peripherals cannot handle complete CAN frames out-of-the-box, we also introduce the concept of polyglot frames, which is inspired by the general notion of polyglots that has been applied in other fields (e.g., polyglot programs, files, signals) [2, 6]. This enables us to identify and produce sequences of bits that are compliant with both the CAN protocol and the involved peripherals, and generate data sequences that can be transmitted by a protocol interface and can be interpreted as a valid message by a different one.

We demonstrate the validity of our approach from different perspectives. First, we verify the presence of pin conflicts between the embedded CAN controller and other less constrained peripherals, such as UART, SPI, I2C, and ADC peripherals, in a variety of automotive-graded microcontrollers and the existence of corresponding polyglot frames. Secondly, we demonstrate that the aforementioned peripherals can be practically used to produce and receive complete CAN frames on a real CAN network, keeping up with the speed of the modern CAN bus. This shows the flexibility of our approach and demonstrates practically that real CAN hardware cannot distinguish CANflict bits from legitimate CAN frames.

Finally, we implement an end-to-end targeted denial-of-service attack using CANflict, showing how a remotely compromised ECU can completely shut down another ECU in the same CAN network without any assumption on the periodicity of the victim's messages, which is instead a requirement for [21].

To summarize, our contributions are the following:

- We present a novel, software-only approach to reliably and precisely read and inject bits on the CAN bus, bypassing the restrictions imposed by the CAN controller;
- We show that this approach can be used to mount link-layer attacks on CAN networks from remotely compromised ECUs, which makes remote link-layer attacks practical;
- We demonstrate the possibility of producing full CAN frames that are completely compliant with CAN timing and format specifications, using polyglot frames;
- Finally, we release[1] an extensible framework that can be used to read and write arbitrary bits on the CAN bus using different microcontrollers and peripherals, and can be further extended to include other hardware and protocols for future research on the topic.

2 CAN PROTOCOL PRIMER

CAN is a bus standard widely used in the automotive industry. The ISO 11898 standard [16] defines three layers for the CAN protocol stack in relation to the OSI model: the *physical layer*, which defines the electrical properties of the bus, the *data link layer*, which defines frame formats, arbitration, and error reporting mechanisms, and the *application layer*, in which further protocols can define their message formats. Each message at the application layer is composed of a payload, which can be `8` bytes long at most in standard CAN, and an ID, which is used as a message identifier and, implicitly, as a priority tag as well, as explained below.

Nodes Layout. At a physical level, CAN is a two-wire differential bus that interconnects nodes in a broadcast fashion. Each node that participates in CAN communication requires a CAN interface, which is composed of a *CAN controller* and a *CAN transceiver*. The *CAN controller* unit can be found as a stand-alone circuit or, more often, as a dedicated module of the host microcontroller. The controller implements the CAN protocol at the data link layer as described by the standard, generating the bit sequence that has to be transmitted on the bus and decoding incoming bits into application-level messages. The *CAN transceiver* is responsible for converting between logical data, coming out and going to the CAN controller, and the corresponding physical signaling, as it connects the CAN controller to the physical communication lines.

It is important to note the difference in the signals handled by these two components: the CAN controller is in charge of converting application-level objects (messages) to a sequence of bits and vice-versa, using the `CANTX` and `CANRX` lines (which are digital). On the other hand, CAN transceivers take care of transforming each bit received from `CANTX` into a voltage difference between `CANH` and `CANL`, and continuously monitor the bus to output the current differential level as a `1` or `0` on the `CANRX` line.

Bits Representation and Message Arbitration. The CAN specification defines two kinds of bit: *dominant* bits, whose logical value is conventionally `0`, and *recessive* bits, corresponding to logical `1`. According to the specification, a dominant bus level must always overwrite a recessive bus level. Therefore, the CAN bus is implemented as a wired-AND bus. As a result, if any device on the CAN bus transmits a dominant bit, which is represented by a `0` at the logical level, it will overwrite any other ongoing communication. This behavior enables the implementation of CAN *arbitration mechanism.* The CAN arbitration mechanism is applied over the first part of the frame, i.e., the ID field. If two or more nodes start sending a frame at the same time, they each continue the transmission as long as the value of the bit read out from the bus equals the value they have written on the bus. Whenever one of the devices reads a value on the bus that is different from the one it has written, it will immediately back off. Since a node putting a recessive bit on the bus will always lose arbitration to a node writing a dominant bit, identifiers with lower values have higher priorities.

Error Handling. The CAN protocol defines an error detection mechanism based on bus monitoring, performed by both the sender and receiver of a message. The sender is responsible for monitoring the sent message bit-by-bit and reading the acknowledge field. Whenever an error is detected, the detecting node starts sending an error frame beginning from the first bit following the error detection. After the error frame is sent and the intermission time has elapsed, the sender of the erroneous message will try to retransmit it. Note that this mechanism happens at the data-link layer: the CAN controller automatically generates error messages and retransmits them after an error has been detected. Therefore, the application layer is never involved in these operations.

[1]Available at https://github.com/necst/CANflict, uploaded for review in the additional materials.

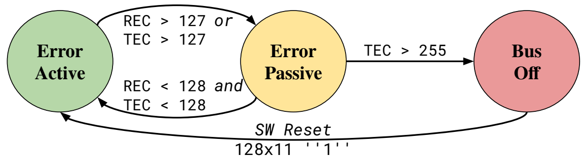

Figure 1: Error states of the CAN bus.

Fault Confinement. The CAN protocol specification describes a fault confinement mechanism to prevent faulty nodes from creating high bus loads. According to this mechanism, each node should implement two error counters: TEC (Transmission Error Counter) and REC (Receive Error Counter). These error counters are decremented by 1 on each successful transmission or reception of a data frame, respectively. Upon detecting an error, the sender node increments TEC by 8, while receivers increment REC by 1 unless they are the ones causing the error, in which case REC is incremented by 8. Depending on the values of these error counters, a CAN node can be in one of three error states:
Error Active: when in this state, the CAN node behaves normally without any specific restriction.
Error Passive: nodes in this state can only indicate an error by sending 6 recessive bits, preventing other nodes from globalizing the error. When sending consecutive data frames, nodes in this error state must wait for an additional time equivalent to 8 bits (Suspended Transmission Time).
Bus-Off: nodes that reach the bus-off state can no longer influence the bus communication in any way. This state can only be exited after 128×11 correctly recorded recessive bits.

Figure 1 shows the possible transitions between these three states along with the triggering conditions.

3 CURRENT STATE OF ATTACKS ON CAN

Like many older low-level protocols, the CAN protocol lacks some fundamental security mechanisms, making vehicles vulnerable to malicious adversaries. As described in [41], CAN main security shortcomings are related to the lack of authentication and encryption, the broadcast transmission, the priority-based arbitration, and the limited bandwidth and payload. On top of the intrinsic security shortcomings of CAN, the environment in which it is most commonly implemented, vehicles, has nowadays multiple attack surfaces, represented by external and internal communication interfaces. In fact, a considerable amount of research has been carried out on this subject [4, 5, 8, 19, 26, 27], highlighting many aspects of modern vehicles that can be exploited by malicious actors. From a general perspective, it is possible to categorize CAN attacks depending on the attacker's location or depending on the network layer at which the attack is carried out. Regarding the attacker location, the most common ways to gain access to the CAN bus are: *local* - having a malicious node physically installed in the electronic system or attached to the diagnostic port by an adversary, and *remote* - remotely compromising a legitimate internal node of the CAN bus. It is evident that the scale of vulnerable targets in the event of an attack that falls in the second category is much greater, as studied by Miller and Valasek in [28], where they observed that many hundreds of thousands of vehicles in the united states were vulnerable to their attack at the time of development. From a network layer perspective, CAN attacks can be carried out at the *application layer* or *data link layer*.

3.1 Application Layer Attacks

In standard CAN networks, an adversary who is able to attach a malicious ECU to the network or reprogram an existing one can typically send and receive messages without any limitations regarding their ID or payload. This capability makes the following attacks possible: **(a)** Eavesdrop Attack - since CAN does not implement encryption and has limited payload and bandwidth, it is uncommon for messages to be fully encrypted, which implies that anybody listening on the bus has full read access to the messages sent by all nodes. **(b)** Spoofing and Replay Attacks - since there is no authentication, an attacker capable of writing a message can impersonate any node, either by forging ID and payload or re-transmitting a previously received message. **(c)** Network Denial of Service - since CAN arbitration depends on the ID, an attacker can overload the bus with 0x00 ID packets, forcing nodes to delay communication.

These attacks can be used to target safety-related systems, e.g., the ABS, change the information displayed on the dashboard, either hiding an existing issue to the driver or signaling a nonexistent one, disturb or take control over autonomous features, such as parking assistance or cruise control, and finally completely shut down a car, as famously demonstrated in [28]. However, since active attacks of this type rely on injecting additional messages on the bus and the vast majority of CAN frames are sent with some degree of periodicity [40], such attacks are trivially detected by modern IDSs [1, 14, 24].

3.2 Data Link Layer Attacks

More recent research has shown that attacks on CAN networks can also be carried out at the data link layer, flying under the radar of message-level IDSs. Some of the attacks that can be carried out at this level are: **(a)** Complete Denial of Service - an electrical property of the CAN bus is that dominant bits, i.e., 0s, have a priority over recessive bits. This means that keeping the bus constantly in a dominant state will prevent any further communication from being performed on the bus. **(b)** Selective Arbitration Denial - since messages with lower IDs have a higher priority in the CAN protocol, injecting dominant bits on the bus while a message ID is being communicated will cause the transmitting device to lose the arbitration, which forces it to back off and stop transmitting. This can be done repeatedly during the transmission of specific messages to prevent an ECU from ever winning bus contention. **(c)** Targeted Denial of Service - if, instead, a dominant bit is injected in the payload of a message, while a transmitting device is sending a recessive bit, the transmitting device will detect an error on the bus, increasing its internal error counter and immediately terminating the transmission. Repeating this process a certain number of times will cause the device to accumulate too many errors, which forces it to go into a bus-off state. This mechanism can be used to completely shut down the communication of any of the nodes connected to the CAN bus. **(d)** Synchronization Disruption - finally, both the synchronization mechanisms and the sampling point settings of the CAN protocol can be used to cause a desynchronization between nodes on the CAN bus, and, in some cases [38, 39], this can cause different nodes to read differently the same message on the bus. This can be particularly useful to evade IDS message inspection or disrupt communication between nodes on the CAN bus.

While complete DoS attacks are trivial to identify and prevent, more sophisticated attacks such as targeted DoS are extremely hard to distinguish from a genuine fault on the bus and are much more challenging to detect with an IDS. However, these attacks come with stronger requirements on the attacker's side. Cho et al.'s approach [9] for example, which is based on overlapping a valid message with a forged one to trigger the generation of errors, requires the attacker to be able to predict the arrival of a message with an error of a fraction of the bit time, and recent work [20] demonstrated its unreliability in real-world scenarios. Attacks such as those introduced by Palanca et al. [32] and further developed in [3, 29] rely on the ability of the attacker to read the initial part of an incoming message and replace a single recessive bit of the payload with a dominant one, which imposes tight timing constraints, especially at high bitrates. Desynchronization attacks have even stricter timing requirements: in [39] the attacker has to craft packets in such a way that the rising or falling edge between one bit and the subsequent ones in the CAN frame happen in a time window that is the order of 1/10 of the bit time (which is already 1μs on a full-speed CAN bus). Such hard timing requirements make implementing these attacks on real microcontrollers, such as those found at the core of automotive ECUs, quite challenging and, in some cases, impossible: in [39], for instance, the experimental evaluation is carried out using an FPGA rather than a microcontroller, while [32] uses a 50kbit/s bus as a target for its attack, which is many times slower than the average. A more extensive evaluation of the performance needed for link-layer attacks, such as [29], shows that the 1μs bit time constraint significantly limits the freedom of the attacker.

3.3 Existing Bit Injection Techniques

The growing interest in low-level CAN bus manipulation in recent years, especially in the car hacking community, has produced a number of tools that can be used to intercept and produce CAN traffic. In particular, open-source tools such as CANT [7] and CANhack [37] use bitbanging to access the CAN bus data-link layer. However, CANT is specifically designed for a high-end microcontroller (STM Nucleo-H743ZI2 clocked at 400MHz) and comes with a custom external shield. CANhack, on the other hand, is a MicroPython tool implemented for both the STM32F405 and the Raspberry Pi Pico, but the tool's author explicitly states that the hardware platform in use must be "fast enough to bit-bang CAN," which excludes lower-end microcontrollers. Finally, the tool has been tested only on a 500kbit/s CAN bus.

More recently, CANnon's approach [21], based on gating the peripheral clock of the CAN controller to delay the sending of a dominant bit until the victim frame is transmitted, significantly relaxes the performance requirements of the targeted platform since any microcontroller with an embedded CAN peripheral can inject bits using this technique. However, this approach has a significant limitation since it does not provide a low-level read primitive. As a matter of fact, while the clock is held in the "loading" phase of the attack, the attacker has no feedback on the current state of the bus. This means that the attacker must rely on periodic messages to know when to "fire" the attack and is blind to the traffic that is happening in real-time on the bus. Hence, Targeted DoS attacks mounted with this technique must make heavy assumptions about the current state of the bus before being triggered and rely on specific characteristics of the CAN network traffic, requiring the attacker to predict the time of arrival of a given CAN frame beforehand, which is not always possible nor practical in real-world scenarios. Finally, this approach is inherently noisy, and the technique discussed by the authors to increase its reliability has the drawback of making it easier to detect for an IDS since it requires to hold a dominant state for a prolonged period of time.

4 CANFLICT: POLYGLOT FRAMES ON CONFLICTING PERIPHERALS

Current state-of-the-art tools and techniques used for CAN bit injection are not suitable to make sophisticated data link layer attacks practical on remotely compromised ECUs. They are either too imprecise to handle the timing constraints required by advanced data-link layer attacks, or they require high-end hardware, which might not always be available or exploitable in real in-vehicle networks, or make strong assumptions on specific characteristics of the CAN bus traffic, which are not always verified in practice. In this paper, we present a novel, flexible approach to reliably access the CAN data link layer from software. Our approach can be used even on low-tier microcontrollers without any additional hardware. Moreover, our approach does not make assumptions on the targeted CAN traffic since it provides robust read and write primitives that can be combined to mount current and future attacks to the CAN data link layer from unmodified, remotely compromised ECUs. The added requirement for our approach is that the pins that connect the CAN peripheral to the bus must be accessible by other peripherals on the same microcontroller. At first glance, this requirement may look limiting, however, this condition is met by the vast majority of modern microcontrollers for the reasons that are explained in Section 4.1. For instance, Table 1 shows a list of conflicts between some of the most common low-level peripherals and the CAN controller on some popular microcontrollers manufactured by the top vendors in the automotive industry.

4.1 Pin Conflicts

At the heart of automotive ECUs, similarly to any other embedded system, we find *Microcontroller Units (MCUs)*. Differently from microprocessors, MCUs are cheap, small, low-power, and specialized computing devices that come with simpler CPUs, on-chip memory, and many hardware peripherals, all baked in the same silicon die.

Table 1: A list of conflicts with CAN peripherals found in popular automotive microcontrollers [12].

Microcontroller	Vendor	# CAN Devices	Conflicts
V850ES/JC3-H	Renesas	1	UART, I2C, GPIO
MPC5554	NXP	3	SPI, GPIO
AT90CAN32	Atmel	1	Timer, GPIO
SPC564A80B4	ST Microelectronics	3	SPI, eSCI, GPIO
C8051F50x	Silicon Labs	1	SPI, I2C, LIN, GPIO
AURIX TC399XP	Infineon	4	SPI, UART, I2C, ADC, GPIO
STM32L562	ST Microelectronics	1	SPI, UART, I2C, GPIO

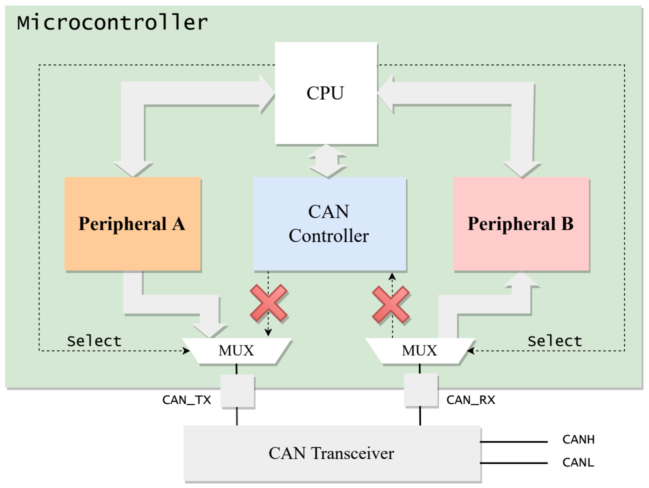

Figure 2: Conflicting peripheral approach to bypass the CAN controller.

They are also typically very limited in memory size and performance: a typical MCU's clock frequency can range from as little as tens of kHz to a few hundred MHz on high-end devices. On the contrary, consumer electronics CPUs nowadays are typically clocked at a speed of several GHz. To cope with the need for real-time responsiveness even with such limited hardware, microcontrollers are equipped with a variety of on-chip *peripherals*, which are designed to efficiently implement some specific, commonly-needed functionality, such as handling SPI or I2C packets, converting analog signals at a high frequency, or reacting to changes in external signals. The growing need for faster communication and the increased number of complex protocols to handle, along with the desire of vendors to provide general-purpose products, has caused modern MCUs to be packed with many of such hardware peripherals, which need more hardware pins than there are available on the physical package. Moreover, each peripheral is typically connected to multiple sets of pins to provide maximum flexibility to customers during the PCB design and routing phases. As a consequence, many peripherals end up sharing the same pins, whose physical connections are internally multiplexed and redirected to the chosen peripheral through a set of *memory mapped registers*, which can be read and written by software. We refer to these pins as "conflicting pins" and the involved peripherals as "conflicting peripherals".

4.2 Bypassing the CAN Controller

As explained in Section 2, nodes in a CAN network access the bus through two components: a controller, which handles the data link layer, and a transceiver, which handles the physical layer. To perform a link-layer attack, we need to access the CAN data link layer, which, however, is hindered by the CAN controller. Indeed, through the CAN controller, it is not possible to directly handle each bit that is sent or received on the bus, nor is it possible to force the timing of reading and writing events from software. As a matter of fact, the software layer can only communicate to the CAN controller the ID and payload of the message it wants to send or get notified when a complete message is received without errors on the bus, while all the bus arbitration, error handling and frame crafting logic are handled automatically by the hardware. This makes link-layer attacks on CAN not feasible through the CAN controller. This may create a false sense of security at design time, built upon the assumption that an attacker cannot access the data-link layer of the protocol. However, CAN controllers are typically found as embedded peripherals in modern MCUs, which means that the same pins that are used by the CAN controller, `CANTX` and `CANRX`, are also shared with other peripherals, as discussed in 4.1.

One way we could access these pins is through the General Purpose Input/Output (GPIO) peripheral, which enables the software to control and read the logical level of a pin through memory-mapped registers. This technique is commonly called *bitbanging* and can be used on high-end microcontrollers to access a relatively low-speed bus. The main limitation of this technique is that a read or write event for each bit must be commanded by the software, which, in the case of a full-speed CAN bus, leaves the CPU with a 1μs window to execute whatever logic is needed for each bit, e.g., saving the current level of the bus in a given sequence, counting how many bits have been received and comparing the received sequence with a given one. This is clearly a strong timing limitation for a microcontroller with a clock of, for example, 10MHz. Moreover, with this technique, the CPU is fully occupied by the reading and writing operations, and any event that can alter the timing of an instruction, such as an Interrupt Request (IRQ), might cause the microcontroller to desynchronize with the bus and the attack to fail or be triggered at the wrong moment. Finally, on a practical side, bitbanging techniques rely either on the presence of a high-resolution timer or on platform-specific fine tuning that might not always be practical and is highly prone to problems such as clock drifts.

Instead, CANflict uses low-level protocol peripherals to control the `CANRX` and `CANTX` pins. Our intuition is that we can leverage the presence of conflicting peripherals on those pins to bypass the CAN peripheral and gain full access to the CAN data-link layer from software, as depicted in Figure 2. In particular, packets in low-level protocols such as Serial Peripheral Interface (SPI) and Universal Asynchronous Receiver Transmitter (UART) have much fewer restrictions than CAN frames and can be concatenated to overlap partially or completely a given CAN message. In this way, we can benefit from the speed and the asynchronous nature of dedicated peripherals to offload the handling of high-speed communications from the CPU, which by itself would not be able to cope with the tight timings imposed by CAN, while still obtaining a high degree of control over the traffic on the bus. The principle described above extends the threat model that car manufacturers should consider while designing vehicular on-board networks by adding this additional attack surface (i.e., remote data-link layer attacks from the application level) that is often neglected in existing threat models [1, 24].

4.3 Polyglot Frames

Bypassing the CAN controller is not, by itself, sufficient to mount sophisticated link-layer attacks since, in order to read and write long sequences of bits, we need to handle CAN data from within another peripheral. For this reason, we introduce the concept of *polyglot frames*. In general, the meaning of a signal is not intrinsic to the signal itself (both for digital, as in our case, or analog ones). A digital signal is merely a temporal sequence of high and low voltages, while the actual information stored in the signal depends on the way it is interpreted, following rules and conventions. Therefore, its meaning is not bound to the physical signal itself but to the interpreter that attributes meaning to it. For example, a sequence of bytes may output a melody if interpreted as an audio track, but it may look random and meaningless if interpreted as an image. Instead, a sequence of bytes valid both as a sound and an image file is referred to as a *Polyglot file* [2, 33]. Following the same reasoning, a bitstream on a communication channel (i.e., frame) transmitted by a protocol interface that is both a valid SPI message and a valid CAN message would be referred to as a *polyglot frame*.

In practice, we use this concept to identify pieces of CAN frames that are compatible with a given protocol, which can then be read and written with the corresponding peripheral. We also leverage the existence of full-frame polyglots between the protocols chosen in this paper and the CAN protocol to demonstrate that CANflict can also generate complete and valid CAN frames that are accepted as such by legitimate nodes.

4.4 Exploiting Conflicting Peripherals

Depending on the peripheral that conflicts with the CAN controller's `RX` and `TX` pins, the capabilities of CANflict vary. Each protocol has its own requirements, and since it is necessary to fulfill them to read or send a CAN frame, this may limit or fully enable the attacker's capabilities. We proceed to analyze the capabilities of our intuition on some of the most common peripherals found on modern microcontrollers, i.e., SPI, UART, I2C, and ADC.

Serial Peripheral Interface (SPI). SPI is a primary-secondary[2] serial protocol that typically employs four lines (See Table 2). Figure 3 represents the timing diagram of a typical SPI communication. When in primary mode, the SPI device automatically selects the proper secondary, generates the clock signal, sends bits on the `COPI` line, and reads bits on the `CIPO` line. Communication from secondary devices to the primary device is also initiated by the primary, which decides which device can communicate on the `CIPO` line by setting the *CS* line of the corresponding secondary low and generating the clock signal. The protocol does not define any intrinsic limitation on the shape of the packets sent and received by the primary. Finally, SPI devices typically expose some mechanism to modify the *Clock Polarity* and *Clock Phase* of the signal, which determine how the bits are encoded during the communication. In particular, the *Clock Polarity* affects the logic level of the CLK signal when the peripheral is idle, while the *Clock Phase* defines whether the read/write operation for each bit begins on the rising or falling edges of each clock pulse.

SPI Polyglot Frames: The main requirement to read on the bus is that the `CIPO` line of the SPI peripheral and the `CANRX` line of the CAN peripheral share the same pin so that the incoming signals from the CAN bus can be redirected to the SPI peripheral. An additional requirement is the knowledge of the baudrate of the CAN bus. Given these two requirements, it is trivial to read bits on the bus. Typically, to read a complete CAN message from the start, the application code needs to recognize the Start of Frame (SoF) bit. Similarly, to write on the bus from the SPI peripheral, the `COPI` line of the SPI peripheral and the `CANTX` line of the CAN peripheral need to share the same pin, alongside the knowledge of the baudrate. Since no specific rules apply when transmitting SPI packets from a primary device, any bitstream that is provided to the peripheral will be transmitted as-is on the `COPI` line. This means that if the `COPI` line conflicts with the `CANTX` signal coming out from the microcontroller, the SPI peripheral can be used to send an arbitrary number of bits of the bus. Note that, since the `CS` line and the `CLK` signal are generated automatically by the SPI peripheral, we can completely ignore their presence. In fact, neither of these signals are relevant for sending arbitrary bits on the CAN bus, and the SPI device never reads them. Figure 6 shows an example of how a sequence of bits transmitted by the SPI peripheral can be interpreted both as a CAN message and an SPI message.

[2] Although the OSHWA suggests [31] *Controller* and *Peripheral* as the new naming convention, we avoid the use of this terminology because it could lead to confusion in our context of microcontrollers. Hence, we will refer to the controller device as *primary* and to the controlled device as *secondary* throughout the text. We will, however, use the suggested acronyms CIPO and COPI in lieu of MISO and MOSI for ease of reference.

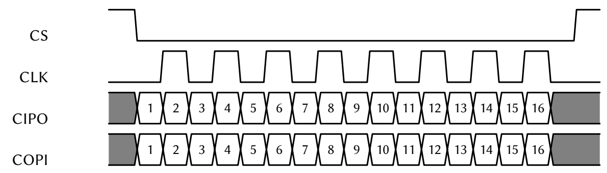

Figure 3: Timing diagram of an SPI message.

Universal Asynchronous Receiver Transmitter (UART). The UART protocol is another widespread serial protocol used in many embedded applications. Unlike SPI, the UART protocol does not use a clock signal to synchronize the transmitter and receiver devices; instead, it transmits data asynchronously. Similar to the CAN peripheral signals, the two main signals of a UART peripheral are the transmission line (*TX*) and receiving line (*RX*). In the UART protocol, each packet must have a predefined form, which consists of a start bit, data frame, a parity bit, and stop bits, as summarized in Table 3 and shown in Figure 4. In particular, the start bit's value is always `0`, the stop bit's value is `1`, and the payload of each packet can contain 5 to 9 bits.

UART Polyglot Frames: If there is a pin conflict between the `CANTX` line and the UART peripheral's `TX` line, we can inject bits on the

Table 2: SPI Lines Description.

Name	Description
CS	Chip select line used by the primary to select which secondary to communicate with.
CLK	Clock signal generated by the primary and sets the bit timing of the communication.
CIPO[2]	Data from secondary to primary.
COPI[2]	Data from primary to secondary.

Table 3: UART frame Description.

Name	Description
Start Bit	Always set to `0`.
Data Frame	Payload can be from 5 to 9 bits long.
Parity Bit	Optional, used for error detection.
Stop Bit(s)	One or two consecutive logical `1`s, depending on peripheral configuration.

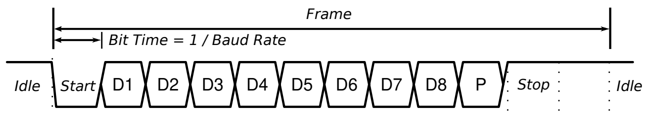

Figure 4: Timing diagram of a UART message.

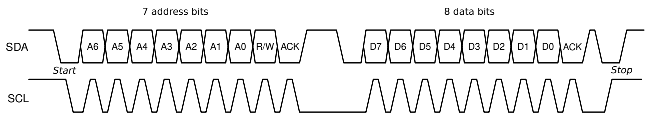

Figure 5: Timing diagram of an I2C message.

CAN bus using the UART peripheral. Equivalently, a pin conflict between the CANRX line and the UART RX line can be exploited to read bits on the bus. Although the UART peripheral still enables an attacker to send and receive bits on the bus at arbitrary moments, bypassing the arbitration logic, the fact that UART packets have fixed start and stop bits imposes some additional constraints with respect to SPI when trying to emulate CAN traffic. In particular, while the payload of each frame is entirely controlled by software, start and stop bits in each packet have fixed values. However, by modifying the packet length, the user can still control the position of these fixed values, which significantly relaxes the practical constraints of emulating and intercepting CAN traffic. In practice, as shown in Section 5, by using a chain of UART packets, it is still possible to match a significant portion of a CAN message, and, in some cases, we can even craft complete CAN frames by just concatenating a sequence of UART packets. Figure 6 shows a particular instance of a UART polyglot frame, in which the same signal can be interpreted as both a CAN frame and a sequence of UART packets. We leverage this property in our experimental validation, proving that a real UART peripheral can generate a signal that is accepted and acknowledged by an unmodified CAN controller.

Given a CAN frame that we want to produce, to obtain a valid UART polyglot, we follow a "greedy" approach: **(a)** Assign the first packet's length, verifying that the last bit is compliant with the stop bit value. **(b)** If this is not the case, modify the length until the first packet is a valid UART packet. **(c)** Verify that the following bit is compliant with the start bit value. If this is not the case, repeat from *b.* until this condition is met. **(d)** Repeat from *a.* for the next UART packet, until all the CAN frame has been covered. **(e)** If no correct solution can be found, backtrack to reassign earlier packets to a different length. Additionally, the stop bit length of each packet can be modified to provide even more solutions.

For what concerns reading, it should be noted that the presence of a start bit in the UART protocol, which is needed to synchronize the communication in the absence of a separate clock line, is actually an advantage for our purpose since it has the same value of the CAN frame SoF bit. This means that the UART peripheral can be configured to start reading the bus at any moment of the inter-frame time window, and it will automatically recognize the start of the next CAN frame. As we already noted, subsequent bits must be compliant with the UART protocol: the next 5 to 9 bits, depending on how the peripheral is configured, are read normally by the peripheral. The following 1 or 2 bits, depending on Stop Bit(s) configuration, must be 1. If not, the UART peripheral discards the incoming message as faulty. The next bit must be a 0 (Start Bit), which forces the peripheral to listen for a new packet, and so on. This implies that, in order to configure the UART peripheral correctly, the attacker must know in advance (at least part of) the sequence of bits that it wants to target.

Inter-Integrated Circuit (I2C). Another popular communication protocol in modern embedded systems is I2C, which is a multi-primary, multi-secondary[2], synchronous protocol. Since it is a serial protocol, data is transferred bit by bit along a single wire, called the SDA line. Like SPI, I2C is synchronous. Hence, the emission and sampling of bits are synchronized by a clock signal shared between the primary and the secondary. The primary controls the clock signal. In the I2C protocol, messages are broken up into two types of frames: an address frame, where the primary indicates the peripheral to which the the message is being sent, and one or more data frames, which are 8-bit data messages passed from primary to peripheral or vice versa. The two lines are called *Serial Data* (SDA) and *Serial Clock* (SCL). Data is placed on the SDA line after SCL goes low and is sampled after the SCL line goes high. The time between clock edge and data read/write is defined by the devices on the bus and varies from chip to chip. The timing diagram of a typical I2C communication is provided in Figure 5. I2C packets are structured as in Table 4.

I2C Polyglot Frames: If a conflict between the *SDA* line and the CANTX line is present in the target microcontroller, this protocol can be used to send bits on the CAN bus. The same does not hold for reading since, in this protocol, the primary must always send a command on the bus before reading. The protocol rules are even more restrictive than UART since the beginning and end of each packet are signaled by a low voltage on the *SDA* line, whose duration depends on the specific device. Moreover, each packet sent by the primary is expected to be acknowledged by the secondary in the *ACK slot*, during which the primary leaves the *SDA* line in a high voltage state. Finally, each frame in the I2C protocol is formed by multiple packets and starts with an *address* packet, in which the primary communicates the address of the secondary, the direction of communication (read or write), and waits for the ACK signal from the secondary. If the message is not acknowledged, the communication is interrupted by the primary. All of these aspects of the protocol interfere with the necessity of sending arbitrary bits on the bus since the voltage of the bus during the start condition, stop condition, ack slot, and inter-frame space cannot be controlled. Nevertheless, as for UART packets, being able to control at least some of the bits that are sent on the bus can be enough to inject

Table 4: I2C communication process.

Name	Description
Start Condition	The SDA line is pulled low while SCL is high to indicate the beginning of communication.
Payload	8 controllable bits on the SDA line, both for address and data frames.
ACK Slot	The SDA line is held high by the primary, and the secondary is expected to pull low (0) the clock for a positive acknowledgment.
Stop Condition	The SDA line is pulled high while SCL is high to indicate the end of communication.

small sequences of bits, and, with enough knowledge of the I2C device characteristics, it is even possible to craft complete, valid CAN messages, as shown in Figure 6.

Similarly to UART, the following steps produce a frame valid both for CAN and I2C: **(a)** Obtain the durations of the I2C start condition, stop condition, ACK slot, and inter-frame space, either from the datasheet or by direct measurement: these are the fixed portions of the I2C frame. **(b)** Verify that the target CAN bit time divides the duration of all fixed portions. This induces a bit representation for each fixed portion. **(c)** Define the number of I2C frames to send. **(d)** Assign the bits corresponding to the I2C fixed portions, as determined in *b.*. **(e)** Assign the bits that have a fixed value in the CAN protocol (e.g., ACK delimiter). **(f)** Choose the remaining bits of the CAN frame, excluding the CRC portion. **(g)** Calculate the CRC of the frame. **(h)** Check if the CRC is compatible with the already assigned bits of that section of the frame. **(i)** Repeat from *f.* if the CRC is not compatible.

Note that both *b.* and *e.* may be unfeasible. On the one hand, given a CAN baudrate there is no guarantee that the I2C writing technique is suited for crafting messages at that baudrate. On the other hand, given a compatible baudrate, the position and length of the I2C fixed portions could interfere with the CAN control fields, leading to a wrongly formatted message. Nevertheless, since many link-layer attacks do not require the ability to craft complete packets, I2C can still be used to attack the CAN data link layer, as demonstrated in Section 5.4.

Analog-to-Digital Converter (ADC). *ADC*s are the last type of peripherals examined in this section. The capabilities of such peripherals may strongly vary among different platforms and vendors, but the basic idea is that they can be used to perform fast and repeated analog-to-digital conversions without the intervention of the CPU (e.g., to sample the *CANRX* signal as if it was an analog signal). Many microcontrollers include one or more ADC devices as on-chip peripherals. Typically, such devices expose a mechanism to regulate the *resolution* of the conversion, i.e., how small can the difference between two analog values be before they become indistinguishable. Clearly, since we are interested in the digital value of the bus, we can select the lowest possible resolution for these conversions, which typically also means higher sampling frequency, and then compare the result with a constant value, corresponding to half of the full-scale range of the ADC. Analog-to-Digital Converters (ADCs) generally expose a simple interface for sampling analog signals at precise intervals.

ADC Polyglot Frames: Clearly, the only action that can be carried out through an ADC peripheral is reading. Therefore, if the `CANRX` signal conflicts with an analog input of the chosen target, the ADC can be used for sniffing bits on the bus. The implementation is straightforward and, in our case, implies the calling of a function every time a new ADC value is received, which checks whether the converted value is greater or lower than the mid-range value of the ADC. We can either listen on the bus until a given sequence of bits is received or record the bus activity for a fixed number of conversions and store the results in a buffer.

5 EXPERIMENTAL VALIDATION

To validate our approach, we prove that conflicting peripherals can be exploited on real hardware and that they can be used to achieve reliable control of the CAN link layer. In particular, we first demonstrate the practicality of producing CAN polyglots from conflicting peripherals and verify that such polyglots are indistinguishable from legitimate CAN bits by testing the exchange of entire CAN frames between a conflicting peripheral and a real CAN controller and showing that no errors are produced. We also benchmark the maximum speed at which such full-frame polyglots can be produced on high-, mid-, and low-end microcontrollers, proving that CANflict provides reliable, high-speed read and write primitives also on low-end microcontrollers. Additionally, we measure the compatibility of messages coming from real CAN traffic with the constraints described in Section 4 for both the UART and the I2C peripheral. Finally, we demonstrate the possibility of mounting advanced link-layer attacks from a real, unmodified microcontroller with CANflict by implementing a targeted DoS attack on simulated CAN traffic, which was recorded from a real car.

5.1 Experiments Setup

The microcontrollers chosen for our experiments are the NXP LPC11C24 [30], a low-end microcontroller equipped with an ARM Cortex M0 processor, the STM32L562 [25], a mid-range controller based on an ARM Cortex M33 processor running up to 110 MHz[3], and the Infineon AURIX TC399XP [15], a high-end, automotive-grade microcontroller with a 6-core processor from the TriCore family. A comparison between the three microcontrollers is provided in Table 5. The choice of these platforms has a twofold aim. First, we want to demonstrate that the techniques presented in this work are flexible with respect to the specific hardware implementation, making them viable on a huge variety of systems. Secondly, we aim at comparing the capabilities of high-, mid-, and low-end platforms, such as the STM32L562, which is found on mid-range systems, and small and inexpensive microcontrollers, such as the LPC11C24, which can be found on simpler systems. Since they differ in many aspects, including the vendor, CPU architecture, clock speed, peripheral chips, and overall performance, they are a perfect fit for demonstrating the flexibility of our approach.

We aim to evaluate all peripherals' capabilities on all the platforms considered to produce a fair performance evaluation. However, as shown in Table 1, the number of conflicts on CAN peripherals is limited. To solve this issue, we simulate some of the conflicts by wiring together the signals coming out from the peripheral under test to the `CANRX` and `CANTX` signals, which are connected to the CAN peripheral. In other words, we are simulating with external wiring the behavior that is normally displayed by internal signal multiplexing. Even if this is not the hardware setup that we have considered when devising the conflicting peripheral techniques, we are confident that our setup closely mimics a situation in which two peripherals have a pin conflict in the chip without losing the capability of evaluating their relative performances. A complete

[3]In the case of the STM32, since the peripheral frequency is obtained by dividing the system clock by a power of 2 (peripheral prescaler), to get a baudrate compatible with typical CAN speeds (e.g., 1 Mbit/s) it is necessary to decrease the system clock speed from 110 MHz to 64 MHz, which can be divided by a power of 2.

demonstration of the feasibility of such techniques on real pin conflicts is provided in the second experiment in Section 5.4.

5.2 Full Frame Experiments

To evaluate the reliability of our approach on a high-speed CAN bus, we decided to leverage the existence of polyglot frames for all the chosen peripherals. For each platform, we simulate the exchange of complete CAN frames between an attacker node, which uses CANflict , and a victim node, which uses a regular CAN controller. In this way, on the one hand, we can generate an entire CAN frame from a peripheral on the attacker node and verify that the victim node's CAN peripheral accepts it without errors by observing the acknowledgment bit on the bus. On the other hand, we can test that entire CAN frames are correctly read by the chosen peripherals on a high-speed CAN bus by verifying that the received bits correspond to the CAN frame sent by the legitimate node. With this strategy, we ensure that all errors and drifts that might accumulate during the transmission or receiving are still smaller than what can be sensed by an ordinary CAN controller. We believe that, in this way, we can convince the reader of the advantage of using conflicting peripherals over other, less reliable techniques. We also imply that if we can repeatedly produce and receive full frames at high speed without errors, we can do so also for smaller portions of the CAN message, which is the typical requirement for data link layer attacks.

More specifically, in our experiments, we used three different frames, a generic one and two specifically crafted ones for the UART and I2C peripherals. The bit representation for each frame can be found in Figure 6.

Generic Frame: The generic frame sent and received through the SPI interface is a CAN frame with the longest possible payload (8 bytes) and generic content. This frame has also been used to validate the ADC reading. It has no particular features and represents a randomly picked, standard CAN frame.

UART Frame: The CAN frame chosen to test the UART peripheral has been instead crafted to respect both the restrictions of CAN frames and those of the UART protocol, as discussed in Section 4.4. In particular, the chosen frame has a 2-byte payload, and the ID and content of the payload have been chosen in such a way that all the fields, including the CRC field, do not break the UART rules.

I2C Frame: The CAN frame sent by the I2C peripheral is a simple remote frame request with a null payload and an ID of `0x38d`. Similar to the previous frame, its values have been selected to be compliant with the strict rules imposed by the I2C protocol, described in Section 4.4. Figure 6 contains the bit representation of this frame, as well as the interpretation for both the CAN protocol and the I2C protocol. In this figure, each packet's payload is colored in violet, while the start condition duration is represented by the space between the pink circle (S) and the start of the payload. Other fixed bits are the ACK bit (green segment), which has to always be 1 since no I2C secondary is connected to send an acknowledgment, and the stop condition, which is the space between the end of the ACK bit and the yellow circle (P). The inter-frame space, which is the time between the stop condition of a packet and the start condition of the following one, is also fixed and must be 1.

As a baseline for our experiments, we use a custom implementation of the bitbanging technique, specifically tailored for each given platform and message. This has been produced using both high-resolution hardware timers and busy-wait loops on the CPU and optimized manually with many trial-and-error attempts in order to show the maximum performance that can be extracted by the platform's CPU for this task. Since such fine-tuning requires physically measuring the accumulated drift for every bit on the bus with an oscilloscope and manually compensating it in the code, we consider this useful only for benchmarking purposes. In fact, an attacker in a remote scenario would not have access to such detailed information about the bit timings on the bus.

Table 5: Comparison between the LPC11C24, STM32L562, and the TC399XP microcontrollers.

	LPC11C24	STM32L562	TC399XP
Vendor	NXP	ST Microelectronics	Infineon
Architecture	Cortex M0	Cortex M33	Tricore 32-bit
Cores	1	1	6
SRAM Size	8 kB	256 kB	2.9 MB
Flash Size	32 kB	512 kB	16 MB
CAN Peripherals	3	1	3
SPI Peripherals	2	3	6
I2C Peripherals	1	4	2
UART Peripherals	1	3	12
ADC Channels	8	2	12
CPU Speed	50 MHz	110 MHz[3]	300 MHz

5.3 Results

Table 6 reports the maximum bus speed at which we were able to send or receive 100 consecutive CAN frames without any errors on the bus. We observe that SPI and UART peripherals are particularly fit for achieving reliable, high-speed communication on the CAN bus, even on low-end microcontrollers such as the LPC11. I2C peripherals, on the other hand, are less useful in full-frame generation and reception due to the restrictions imposed by the protocol, but they can be nevertheless used to generate shorter sequences, as shown in Section 5.4. Since the AURIX TC399XP and STM32 L562 MCUs are significantly more powerful than the LPC microcontroller, we were able to achieve the highest possible bitrate also with our custom bitbanging implementation. However, even on such platforms, reading bits with bitbanging and busy-wait loops was proven to be less reliable than writing since some packets were incorrectly read at a baudrate of 1 Mbit/s. The main reason is that, while during writing, the transmitting device imposes the timing of the communication to other devices using the *soft resynchronization* mechanism, during reading operations, the device does not have this power. In our case, since the interval generated by software between one bit and the other was slightly more than 1 μs, this difference was accumulated during the sampling of the packet until a bit was incorrectly read. Decreasing the interval between bits caused instead the bit timing to be much lower than 1 μs and bits to be read twice. Using the high-resolution hardware timer to resynchronize the communication periodically was enough to cope with these small timing deviations on the TC399.

On the other hand, in the LPC microcontroller, the difference between bitbanging and our approach is more evident, as the CPU alone cannot cope with the speed of the bus. This is a significant

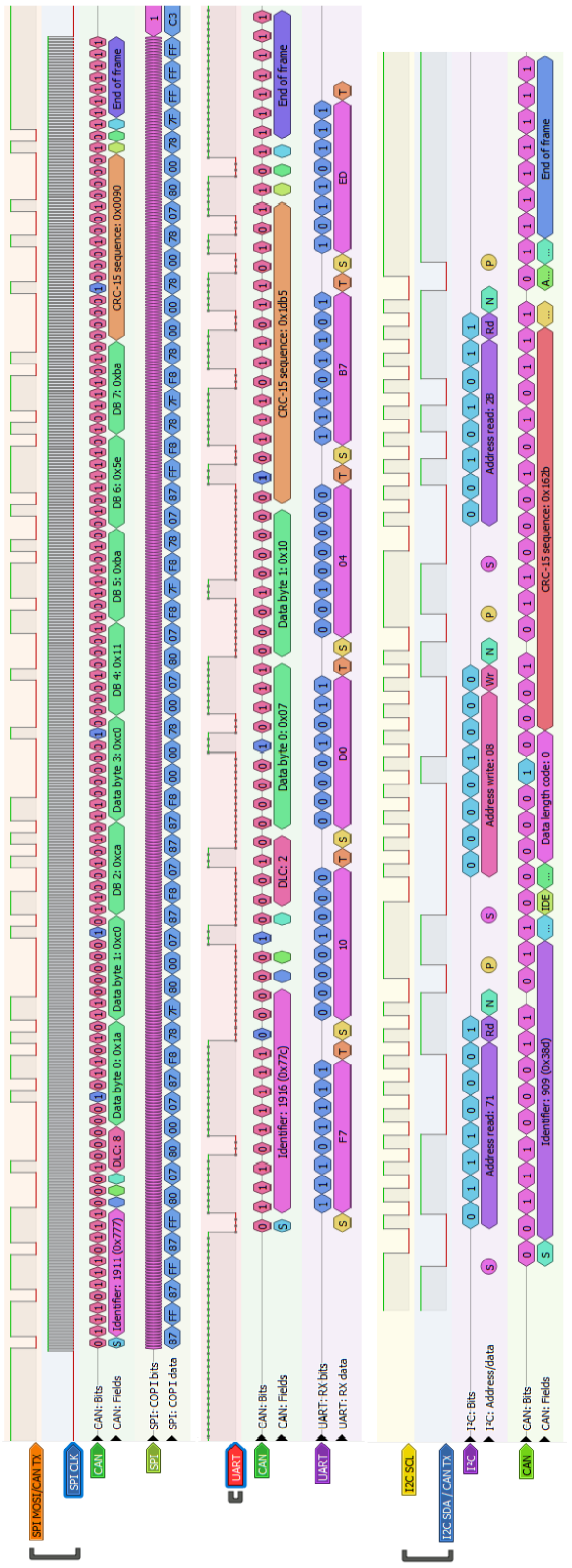

Figure 6: Three examples of how SPI , UART, and I2C peripherals can be employed to craft valid CAN messages.

result since it shows how peripheral-based techniques enable previously impossible precision in injecting and reading bits on the bus. In particular, on this platform, writing techniques that employed hardware peripherals performed 5 to 10 times better than the basic bitbanging implementation, reaching the maximum bus bitrate of 1 Mbit/s, while the bitbanging techniques could not be faster than 200 kbit/s. This demonstrates the increased reliability of our techniques with respect to traditional bitbanging on such platforms.

Finally, due to the restrictions that the protocol imposes, the I2C technique was implemented only for a specific bitrate (200 kbit/s for the LPC and 100 kbit/s for the STM32 platforms). To better explain I2C restrictions, the measured timings for the fixed portions of the frames for the LPC platform were: 5.2 μs (start), 4.41 μs (ack), 5.33 μs (stop), and 9.58 μs (interframe space), with an expected error of ± 0.25 μs. Therefore, a bit time of 5 μs (equivalent to a CAN baudrate of 200 kHz) can contain start, ack, and stop conditions while the interframe space is contained in 2 CAN bits. Empirically, we discovered that at this baudrate (200 kHz), the deviations of I2C from the nominal bit time are acceptable, and the error w.r.t. the nominal bit time is absorbed by the synchronization mechanisms of CAN. This does not hold for multiples of this baudrate since the CAN bit time becomes smaller (e.g., 2.5 μs for a baudrate of 400 kHz), and therefore, the relative error generated by the deviations of the I2C fixed portions increases.

5.4 Targeted Denial of Service Experiments

To demonstrate the capabilities of CANflict in real-world scenarios and its adaptability in terms of possible peripheral combinations, we implement a targeted denial of service attack against a busy CAN network with a baudrate of 1Mb/s, using an SPI peripheral to read and an I2C peripheral to write on the bus.

We choose the STM32 L562 as the attacker, the AURIX TC399 first CAN line as the victim, and its second CAN line as a traffic generator (the AURIX board has two physically separated, completely independent CAN nodes). The traffic chosen to simulate a real vehicle was retrieved by the ReCAN dataset [40]. One of the IDs (`0x1A2`) is chosen to be the victim frame and is transmitted by the victim node, while the rest of the dataset is being transmitted by the traffic generator node. The attacker uses the SPI peripheral, connected to the `CANRX` pin, to scan the bus searching for the victim's ID, and the I2C peripheral, connected to the `CANTX` pin, to write a sequence of dominant bits to trigger the detection of an error by the victim and its fault confinement mechanism. In Figure 7, we see a trace captured while the attacker triggers an error on the bus

Table 6: Performances of peripherals w.r.t each boards of our experiments in write (W) mode or read (R) mode. - refers to cases with impossible setups, while `n.a.` refers to implementations that are theoretically feasible but our framework does not support yet.

Platform	LPC11C24		STM32L562		TC399XP	
	W	R	W	R	W	R
Bitbanging	200 kb/s	120 kb/s	1 Mb/s	500 kb/s	1 Mb/s	1 Mb/s
SPI	1 Mb/s	1 Mb/s	1 Mb/s	1 Mb/s	1 Mb/s	1 Mb/s
UART	1 Mb/s	1 Mb/s	1 Mb/s	1 Mb/s	1 Mb/s	1 Mb/s
I2C	200 kb/s	-	100 kb/s	-	n.a.	-
ADC	-	<50 kb/s	-	300 kb/s	-	1 Mb/s

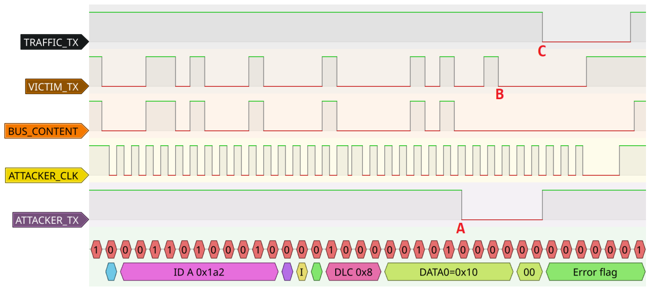

Figure 7: Trace representing an attacker triggering an error on a victim's frame. The attacker is using SPI to read and I2C to write on the bus. The attack starts at A, triggers an error flag by the victim at B, and by the rest of the bus at C.

while the victim is communicating: The attacker forces the bus in the dominant state for the duration of 6 bits (A) while the victim sends its payload, triggering an error flag from the victim (B), and from the rest of the nodes on the bus at (C).

The results of the experiment are optimal: at the highest CAN baudrate of 1Mb/s, the STM32 has no issues detecting the ID of the victim through the SPI peripheral and then generate the error through I2C one. This process is executed 32 times to trigger the fault confinement mechanism of the victim and prevent it from communicating on the bus. The attacker never erroneously generates an error on a frame with another ID. The experiment was repeated multiple times to ensure consistency of the results.

5.5 Polyglot frames compatibility in CAN traffic

Polyglot frames are exploited by CANflict both in reading and writing operations. While writing a polyglot frame leaves room for the attacker to edit the frame to make it compliant with the chosen protocol (i.e., SPI, UART, I2C), the same editing clearly can not be done while reading. To study such compatibility, e.g., for reading other ECUs data, we evaluate the compatibility of polyglot frames between real-world CAN traffic obtained from the ReCAN dataset [40] and constrained protocols [4] (i.e., UART, I2C).

Figure 8 represents how many of the 260k CAN packets in the dataset under analysis are compatible with UART and I2C in terms of the length of consecutive compatible bits from the beginning of the CAN packet. UART, even considering its constraints, has no issues in reading the complete CAN ID and is, therefore, a valid reading technique for many link-layer attacks in case of a pin conflict between it and the CAN RX line. On the other side, I2C, which is capable of reading only the 14.3% of CAN IDs, is less viable to read information from the bus. It is important to note, however, that these numbers are meaningful only in case the attacker is using these peripherals to handle or read fixed CAN data.

In the writing scenario, where the attacker has more control over the message to send, the feasibility of sending full CAN frames, especially with UART, significantly increases.

6 THE CANFLICT FRAMEWORK

As discussed in Sections 4 and 5, exploiting polyglot frames to allow access to link-layer of CAN bus without specialized hardware is the core concept behind CANflict. In particular, link-layer access enables the implementations of attacks, and more importantly, the implementation of defence mechanisms such as CopyCAN [23]. This paper focuses on the feasibility of this intuition on the most commonly found peripherals (SPI, UART, I2C, and ADC), even as the concept can be extended to many others. On top of this, we tested and implemented CANflict on some specific platforms. The implementation on others may require adaptations.

For these reasons we designed the CANflict framework, publicly available online[1], which is designed to group together all the techniques presented in this paper under a common interface. The framework already contains all the code necessary to reproduce the attacks and run our experiments on our chosen platforms. The final goal of the framework is that of providing an extensible, cross-platform environment created by the collective, that can be extended by researchers when using CANflict on novel platforms, while hiding platform specific details such as registers location and peripheral settings. To achieve this goal, the framework is logically split into three different layers:

Public interface Layer. The Public Interface Layer is in charge of defining a unified interface for reading and writing bits on the CAN bus. This interface is meant to be the main point of contact between user code implementing a specific attack and the underlying technique used to mount the attack. More specifically, the reading and writing primitives are provided by two different interfaces: the Sender interface and the Receiver interface. This enables the user to

[4]SPI is not considered in this evaluation since it does not add any requirements regarding the sequence of transmitted bits

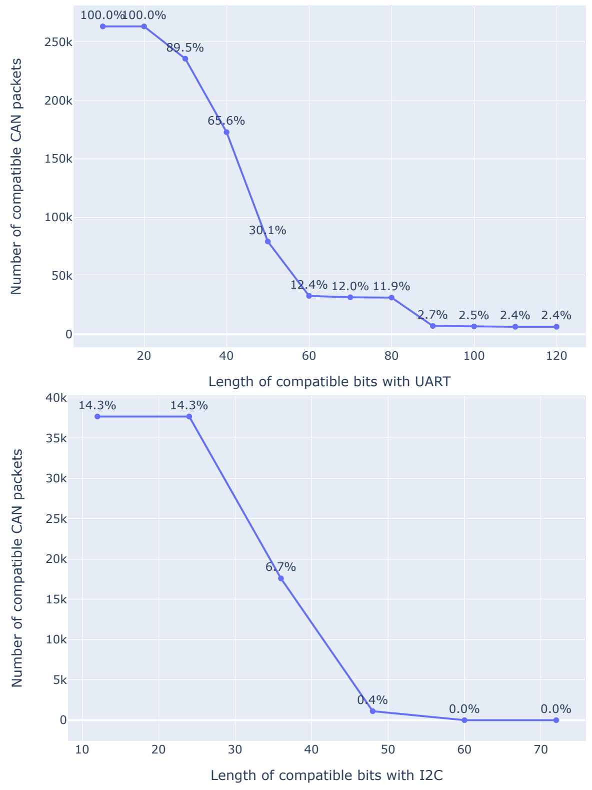

Figure 8: Number of CAN messages taken from a dataset of ReCAN[40] are compatible polyglots with UART (first graph) and I2C (second graph), and for how many bits.

Table 7: A comparison of the techniques that can be used to perform data link layer attacks on the CAN bus

	Cross-Platform	Read	Write	Low-end MCUs	Non-periodic Messages
CANnon [21]	✓		✓	✓	
CANT [7]		✓	✓		✓
CANhack [37]	~	✓	✓		✓
CANflict	✓	✓	✓	✓	✓

use one peripheral for sending and a different peripheral for receiving bits on the bus, which means that it is not necessary to have a full conflict between a peripheral and the CAN controller in order to use its related technique, but also partially overlapping peripherals are allowed.

Techniques Layer. Techniques are divided into Senders and Receivers, each of which implements one of the two public interfaces. Since we do not want to rewrite each technique for each possible platform, the Platform Layer provides a set of abstract peripherals that define a minimal interface to interact with each peripheral. Techniques can use these abstract peripherals to ensure compatibility with all the supported hardware, which completely decouples techniques development from platform-specific code.

Platform Layer. The Platform Layer provides all the code related to the interaction with the hardware. Here, we define some of the most common peripherals, such as I2C, SPI, and GPIO. Each abstract peripheral defines an abstract structure that is implemented by platform-specific code, through which it can be identified and passed to the related functions. The abstract peripherals are then implemented in the Platform code. Platforms are the final targets that will be executing the code (typically microcontrollers). Different vendors provide different functionalities for each of their microcontrollers, so each platform might have a slightly different implementation for each peripheral's functionality.

6.1 Comparison with Exisiting Solutions

Table 7 compares the CANflict framework with existing solutions that perform CAN data-link layer attacks [7, 21, 37]. As discussed in Section 3.3, both CANT and CANhack rely on bitbanging, which makes them impractical on low-end microcontrollers. In addition, the CANT tool is not designed to be cross-platform, while CANhack provides partial portability, although being dependent on the presence of a MicroPython environment. Finally, CANnon, which can be considered the state of the art for performing data-link layer attacks, can be theoretically deployed on any microcontroller with an on-chip CAN controller, but it relies on periodic messages and does not provide reading primitives. On the contrary, the CANflict framework provides both reading and writing primitives allowing deployment on any microcontroller independently from the periodicity of the messages. The only requirement is the presence of a pin conflict, which is however common in ECUs, as shown in Table 1.

7 DETECTION AND PREVENTION OF CANFLICT

The vast majority of the current literature on intrusion detection for CAN focuses on application layer countermeasures [1]. These software countermeasures interface with the CAN controller to receive only the ID and the payload of a correctly received packet. Therefore, they can only process such information. As mentioned in Section 2, the CAN controller does not forward to the microcontroller any information regarding errors and discarded packets, nor a bit-by-bit view of the bus. For this reason, these IDSs cannot detect attacks thoughtfully implemented through CANflict or exploiting link-layer attacks in the literature [3, 9, 21, 29, 32, 39].

Defenses at the data-link layer are much less common on CAN due to the lack of perceived threats. However, it is important to mention CopyCAN [23] and secure CAN transceivers like the NXP TJA115x [13, 34]. The first implements a technique to calculate the transmit error counters of ECUs by reading bus events, therefore detecting an attacker only when it sends spoofed frames after forcing the victim into a bus-off state. The second is a secure CAN transceiver that filters incoming and outgoing CAN frames by their ID. Moreover, these transceivers can act as a tamper protection mechanism and invalidate messages on the bus in case of spoofing. Both are theoretically viable solutions to limit the capabilities of an attacker that exploits CANflict. Furthermore, one of the most significant drawbacks of CopyCAN [23] is the requirement of slow bitbanging techniques to read the bus. It is worth mentioning that our approach significantly lowers computation requirements for link-layer protection mechanisms, thus enabling CopyCAN implementations on low-end hardware.

Finally, countermeasures have also been designed at the physical layer. Multiple works [10, 11] demonstrate the use of voltage fingerprinting of ECUs to recognize spoofing. These techniques are, currently, only partially effective in mitigating CANflict. In fact, since they are designed to recognize spoofing of complete frames, they do not detect shorter injections, which is the only requirement for many link-layer attacks.

CANflict aware security. On top of the already existing security solutions, which can partially limit CANflict capabilities, the most effective countermeasure against CANflict is avoiding pin conflicts with CAN peripherals. We envision this to be obtainable by either choosing an ECU microcontroller with knowledge of CANflict (i.e., choosing hardware that lacks pin conflicts) or by designing the microcontroller itself without conflicts between CAN pins and other peripherals. Similarly, employing an external CAN controller removes the exploitability of pin conflicts, hence securing the platform from CANflict.

8 CONCLUSIONS

In this paper, we presented CANflict, a novel approach that exploits polyglot frames and pin conflicts to perform data-link layer attacks against CAN, making use of different peripherals already present on the microcontroller. CANflict enables an attacker to exploit known vulnerabilities of the CAN protocol to remotely implement read and write attacks without any assumption on traffic periodicity.

We experimentally validated CANflict by studying its feasibility in exploiting existing peripherals protocols, its effectiveness and efficiency in deploying existing attacks, and its compatibility with both low- and high-end microcontrollers on real CAN traffic. First, we demonstrated the feasibility of CANflict on some of the most common peripherals found on standard platforms (i.e., SPI, UART, I2C, and ADC). Then, we verified the effectiveness and efficiency of our intuitions by implementing full CAN communication between one standard CAN node and one where CANflict is implemented, showing that our techniques heavily reduce the computational requirements for link-layer attacks, enabling such attacks from remote on low-end microcontrollers. Moreover, we proved the effectiveness of CANflict even with partial conflicting peripherals by implementing a targeted denial of service attack that uses SPI and I2C to respectively read and write on the bus. Finally, we evaluated the compatibility of polyglot frames between real-world CAN traffic and the UART and I2C protocols. We provide the community the CANflict framework to enable the implementation of our approach on different platforms and peripherals through an easy-to-use and expandable interface. Future works will focus on extending the framework with new platforms and new peripherals, evaluating the feasibility of applying our intuitions on other protocols aside CAN, and in the design of data-link layer countermeasures to detect CAN attacks through the use of CANflict.

ACKNOWLEDGEMENTS

This work was supported by the Dutch Science Organization NWO through project Intersect.

REFERENCES

[1] Omar Y. Al-Jarrah, Carsten Maple, Mehrdad Dianati, David Oxtoby, and Alex Mouzakitis. 2019. Intrusion Detection Systems for Intra-Vehicle Networks: A Review. *IEEE Access* 7 (2019), 21266–21289. https://doi.org/10.1109/ACCESS.2019.2894183

[2] Ange Albertini. 2014. This PDF is a JPEG; or, This Proof of Concept is a Picture of Cats. *PoC or GTFO 0x03* (2014).

[3] Gedare Bloom. 2021. WeepingCAN: A Stealthy CAN Bus-off Attack. *Workshop on Automotive and Autonomous Vehicle Security (AutoSec) 2021* 2021 (02 2021). https://doi.org/10.14722/autosec.2021.23002

[4] Mehmet Bozdal, Mohammad Samie, Sohaib Aslam, and Ian Jennions. 2020. Evaluation of CAN Bus Security Challenges. *Sensors* 20, 8 (2020). https://doi.org/10.3390/s20082364

[5] Mehmet Bozdal, Mohammad Samie, and Ian Jennions. 2018. A Survey on CAN Bus Protocol: Attacks, Challenges, and Potential Solutions. In *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*. 201–205. https://doi.org/10.1109/iCCECOME.2018.8658720

[6] Sergey Bratus, Travis Goodspeed, Ange Albertini, and Debanjum S. Solanky. 2016. Fillory of PHY: Toward a Periodic Table of Signal Corruption Exploits and Polyglots in Digital Radio. In *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016*.

[7] Tim Brom. 2018. *CANT*. https://github.com/bitbane/CANT

[8] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proceedings of the 20th USENIX Conference on Security* (San Francisco, CA) *(SEC'11)*. USA, 6.

[9] Kyong-Tak Cho and Kang G. Shin. 2016. Error Handling of In-Vehicle Networks Makes Them Vulnerable. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. 1044–1055. https://doi.org/10.1145/2976749.2978302

[10] Kyong-Tak Cho and Kang G. Shin. 2017. Viden: Attacker Identification on In-Vehicle Networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. 1109–1123. https://doi.org/10.1145/3133956.3134001

[11] Wonsuk Choi, Kyungho Joo, Hyo Jin Jo, Moon Chan Park, and Dong Hoon Lee. 2018. VoltageIDS: Low-Level Communication Characteristics for Automotive Intrusion Detection System. *IEEE Transactions on Information Forensics and Security* 13, 8 (2018), 2114–2129. https://doi.org/10.1109/TIFS.2018.2812149

[12] John Donovan. 2014. What Engineers Need to Know When Selecting an Automotive-Qualified MCU for Vehicle Applications. https://www.digikey.com/en/articles/what-engineers-need-to-know-when-selecting-an-automotive-qualified-mcu-for-vehicle-applications

[13] Bernd Elend and Tony Adamson. 2017. Cyber security enhancing CAN transceivers. In *Proceedings of the 16th International CAN Conference*.

[14] Markus Hanselmann, Thilo Strauss, Katharina Dormann, and Holger Ulmer. 2020. CANet: An Unsupervised Intrusion Detection System for High Dimensional CAN Bus Data. *IEEE Access* 8 (2020), 58194–58205.

[15] Infineon. 2022. SAK-TC399XP MCU. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/aurix-family-tc39xxx/sak-tc399xp-256f300s-bd/

[16] ISO Central Secretary. 2003. *Road vehicles — Controller area network (CAN) — Part 2: High-speed medium access unit*. Standard ISO 11898-2:2003. International Organization for Standardization, Geneva, CH. https://www.iso.org/standard/33423.html

[17] KeenLab Security. 2018. Experimental Security Assessment of BMW Cars.

[18] Kaveh Bakhsh Kelarestaghi, Mahsa Foruhandeh, Kevin Heaslip, and Ryan Gerdes. 2021. Intelligent Transportation System Security: Impact-Oriented Risk Assessment of in-Vehicle Networks. *IEEE Intelligent Transportation Systems Magazine* 13, 2 (2021), 91–104. https://doi.org/10.1109/MITS.2018.2889714

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

[20] Sekar Kulandaivel, Tushar Goyal, Arnav Kumar Agrawal, and Vyas Sekar. 2019. CANvas: Fast and Inexpensive Automotive Network Mapping. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 389–405.

[21] Sekar Kulandaivel, Shalabh Jain, Jorge Guajardo, and Vyas Sekar. 2021. CANNON: Reliable and Stealthy Remote Shutdown Attacks via Unaltered Automotive Microcontrollers. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 195–210. https://doi.org/10.1109/SP40001.2021.00122

[22] Stefano Longari, Andrea Cannizzo, Michele Carminati, and Stefano Zanero. 2019. A secure-by-design framework for automotive on-board network risk analysis. In *2019 IEEE Vehicular Networking Conference (VNC)*. IEEE, 1–8.

[23] Stefano Longari, Matteo Penco, Michele Carminati, and Stefano Zanero. 2019. CopyCAN: An Error-Handling Protocol based Intrusion Detection System for Controller Area Network. In *Proceedings of the ACM Workshop on Cyber-Physical Systems Security & Privacy*. 39–50.

[24] Stefano Longari, Daniel Humberto Nova Valcarcel, Mattia Zago, Michele Carminati, and Stefano Zanero. 2021. CANnolo: An Anomaly Detection System Based on LSTM Autoencoders for Controller Area Network. *IEEE Trans. Netw. Serv. Manag.* 18, 2 (2021), 1913–1924. https://doi.org/10.1109/TNSM.2020.3038991

[25] ST Microelectronics. 2022. STM32L5x2. https://www.st.com/en/microcontrollers-microprocessors/stm32l5x2.html

[26] Charlie Miller and Chris Valasek. 2013. Adventures in automotive networks and control units. *DEF CON* 21 (2013), 260–264.

[27] Charlie Miller and Chris Valasek. 2014. A survey of remote automotive attack surfaces. *Black Hat USA 2014* (2014).

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

[29] Pal-Stefan Murvay and Bogdan Groza. 2017. 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* (Reggio Calabria, Italy) *(ARES '17)*. Article 71, 10 pages. https://doi.org/10.1145/3098954.3103174

[30] NXP. 2016. LPC11C00 microcontroller family. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc1100-cortex-m0-plus-m0/scalable-entry-level-32-bit-microcontroller-mcu-based-on-arm-cortex-m0-cores:LPC11C00

[31] oshaw 2022. A Resolution to Redefine SPI Signal Names. https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names/.

[32] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. 2017. A Stealth, Selective, Link-Layer Denial-of-Service Attack Against Automotive Networks. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017 (Lecture Notes in Computer Science, Vol. 10327)*. Springer, 185–206. https://doi.org/10.1007/978-3-319-60876-1_9

[33] polyglots 2013. Polyglots PoCs. https://github.com/corkami/pocs.

[34] NXP Semiconductors. 2019. NXP TJA115x Secure CAN Transceiver Family. https://www.nxp.com/docs/en/fact-sheet/SECURCANTRLFUS.pdf

[35] Yuefend Du Sen Nie, Ling Lie. 2017. Free-Fall: Hacking Tesla from Wireless to CAN Bus. In *Black Hat USA 2017*.

[36] IEEE Spectrum. 2020. How Software Is Eating the Car. https://spectrum.ieee.org/software-eating-car

[37] Ken Tindell. 2020. CANhack. https://github.com/kentindell/canhack

[38] Ken Tindell. 2021. The Janus Attack. https://kentindell.github.io/2021/07/15/janus-attack/

[39] Li Yue, Zheming Li, Tingting Yin, and Chao Zhang. 2021. CANCloak: Deceiving Two ECUs with One Frame. *Workshop on Automotive and Autonomous Vehicle Security (AutoSec) 2021* 2021 (02 2021). https://doi.org/10.14722/autosec.2021.23024

[40] Mattia Zago, Stefano Longari, Andrea Tricarico, Michele Carminati, Manuel Gil Pérez, Gregorio Martínez Pérez, and Stefano Zanero. 2020. ReCAN–Dataset for reverse engineering of Controller Area Networks. *Data in brief* 29 (2020), 105149.

[41] Haichun Zhang, Xu Meng, Xiong Zhang, and Zhenglin Liu. 2020. CANsec: A Practical in-Vehicle Controller Area Network Security Evaluation Tool. *Sensors* 20, 17 (2020). https://doi.org/10.3390/s20174900