Metasploit for Cyber-Physical Security Testing with Real-Time Constraints

by

Sulav Lal Shrestha

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2023

Author's Declaration

Statement of Contributions

Chapter 1, Section 2.4 from Chapter 2, Chapter 3, Chapter 4 and Chapter 5 of this thesis have been adapted from the paper [67] which has been published in the SciSec 2022: Science of Cyber Security conference in Matsue, Japan. The portions of the thesis adapted from the paper are reproduced with permission from Springer Nature. The author of this thesis is the primary author of the accepted paper, along with co-authors Taylor Lee and Professor Sebastian Fischmeister. Taylor Lee is a Graduate Student under the supervision of Professor Sebastian Fischmeister in the Real-Time Embedded Software Group at the University of Waterloo.

Abstract

Metasploit is a framework for cybersecurity testing. The Metasploit Framework introduced the Hardware Bridge API to enable security testing of cyber-physical systems. Cyber-physical systems and tests/attacks on the systems are subject to real-time constraints. Hence, this research aims to study the temporal characteristics of tests implemented using the framework. Several factors, such as the programming language used to write tests, overhead added by the framework, scheduling policies, etc., affect the latency and jitter. This study considers the Controller Area Network (CAN) used in automotive systems to study the effect of those factors on the temporal characteristics of the tests. The study evaluates (i) latency and jitter for transmission and reception of the CAN messages in the network and (ii) the jitter in the periodicity in the periodic transmission of CAN messages. Based on the results, the study determines the best combination of the factors to minimize the latency and jitter in the tasks considered.

This work performs a case study on actual tests/attacks subject to real-time constraints and analyses the suitability of executing the tests using Metasploit. The study analyses the performance of tasks implemented as Metasploit modules and shows how choices of some factors can significantly improve the temporal characteristics without modifying the Metasploit Framework. The study compares the temporal characteristics of the tests implemented using the Metasploit Framework to the tests implemented using a microcontroller platform, in this case, Arduino Uno. This work proposes a framework to integrate the Metasploit Framework with tests that are executed on a microcontroller platform.

Acknowledgements

I would like to thank my supervisor Dr. Sebastian Fischmeister for providing me with the opportunity to join the research project and for his guidance in developing the thesis.

Dedication

To my parents.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ANOVA Analysis of Variance 19

CAN Controller Area Network 1–3, 5, 7, 10–16, 21–23, 25, 27–31, 34–36, 39, 45, 46, 48

CLES Common Language Effect Size 20, 34, 48, 50, 52, 54

CPS Cyber-Physical System 1, 2, 45

DoS Denial-of-Service 13

ECU Electronic Control Unit 5, 12

FFI Foreign Function Interface 24

IQR Inter-Quartile Range 48, 49, 51, 53

JIT Just-in-Time 2, 3, 21, 23, 24, 33, 45, 46

REC Reception Error Count 10, 11

RTR Remote Transmission Request 10

TEC Transmission Error Count 10, 11, 14

UART Universal Asynchronous Receiver-Transmitter 44

Chapter 1

Introduction

Integrating networked computing with physical systems exposes the physical systems to threats typically seen in the cyber domain. With advancements in computer-controlled physical systems and networked systems, physical systems are progressively going cyber-physical. A malicious agent can gain control of the networked system or disturb the system, which may lead to disastrous consequences. Stuxnet is one such example of threats to Cyber-Physical System (CPS) that caused sustainable damage to a nuclear program [45].

The automotive industry is one such industry in which traditionally physical components like the powertrain are getting integrated with computing, and more essentially, networked computing [37]. Such integration allows attackers to access the systems inside the vehicle and compromise the subsystems [35]. Once the attackers gain access to some subsystems, the attackers can find ways to access other subsystems. One notable case study was an attack on a 2014 Jeep Cherokee, which found the ability to kill the engine, turn the brakes off and attack the parking assist feature to take control of the steering [54].

Attacks on CPS can have real-time constraints. The Bus-Off attack [24] on the Controller Area Network (CAN) [19] of automotive systems is one such attack with temporal constraints. The correctness of real-time processes depends upon the completion of the process within the given temporal constraints. Deviation from the given timing requirement can cause the system to fail. For example, control algorithms require strict timing and are initially designed without taking into consideration the delays and jitter that exist in real-world scenarios. Consequently, the delays and jitter degrade the performance of the control system [51]. CPS experience similar problems to control systems but face longer delays as network latency amplifies the jitter, which degrades the performance further [15]. In the context of real-time attacks, the effectiveness or correctness of the attack depends on the timely execution of the attack.

For any given task, the use of a framework standardizes the process of executing the task. Such standardization makes the task execution uniform and makes knowledge transfer easier. This study focuses on a framework for cybersecurity: Metasploit [13]. The Metasploit Framework consists of a suite of tools used to test security vulnerabilities and execute attacks. Any task, such as scanning a target or exploiting the target, is implemented as a module in Metasploit. A framework, such as Metasploit, also promotes the reuse of code and components, in this case, the modules. For CPS, it is important to test the security of both the cyber components and cyber-controlled physical components. The Hardware Bridge API in the Metasploit Framework extends the framework's support to hardware-related tests for CPS like Internet-of-Things and Industrial Control Systems [68].

Several factors can affect the temporal characteristics of a process running in an operating system (OS). Such factors include the priority of the process, the scheduling policy used by the OS, and the language used to implement the software. Hence, to make the use of Metasploit suitable for executing real-time tasks, this research studies the temporal characteristics of tests implemented as a Metasploit module. This work considers the following factors (i) the use of a framework as opposed to a standalone program, (ii) the scheduling policy used by the operating system, (iii) process priority, (iv) the programming language used to implement the test (v) use of Just-in-Time (JIT) compilation in case of Ruby programming language and (vi) periodicity in the case of the periodic task. To evaluate the effects of the factors, the study performs three experiments to evaluate (i) message transmission latency, (ii) message round-trip latency and (iii) jitter in periodic message transmission.

1.1 Related Work

Pozzobon et al. evaluated several CAN interface access modules for use in automotive security test frameworks, using round-trip latency and transmission-reception rates as the evaluation metrics [61]. In the study, Pozzobon et al. studied the round-trip latency, transmission rate and reception rate of CAN messages for multiple hardware modules like the Native CAN interface in Beaglebone Black, CAN interface in Raspberry Pi 2 using MCP2515 CAN controller [43], USBtin USB-to-CAN adapter [31], etc. Sojka et al. [69] provided a comparison of Linux CAN drivers - LinCAN and SocketCAN based on the round-trip time of CAN messages.

Researchers have studied the effects of different task compositions, schedules, and priorities [23, 51] on the jitter in real-time tasks. In the case of a General Purpose Operating

System, the user of the OS does not have much control over the scheduling policies. Linux does provide the ability to change the priority and scheduling algorithm for a process to some extent. Dubey et al. [27] implemented a control algorithm to control jitter in the sampling interval for a program running on a General Purpose Operating System. The authors treated jitter in the sampling interval as an error signal that the control system aimed to control. The suggested control algorithm ran in user-space instead of kernel-space, which made the use of the algorithm easier.

When the control task includes nodes in a network, the network latency also acts as a source of jitter. Roque et al. [63] evaluated the performance of an in-vehicle communication system by characterizing the jitter present in the timing of critical CAN messages. Imai et al. [41] used a time-delay compensation method to compensate for jitter in a networked control system. The method uses "Jitter Buffers" to suppress jitter due to time-varying network latency.

Researchers have dedicated efforts to integrate systems with special requirements into general-purpose systems. Bollella and Jeffay [18] designed a system in which a real-time kernel could co-exist with a general-purpose kernel to support real-time solutions in commercial systems. Ramamritham et al. [62] acknowledged that even though Windows NT is not a real-time OS, the timing requirements for the OS warranted a study due to the acceptance of the OS in industrial applications.

The use of Metasploit as a cyber-physical security testing tool for time-sensitive applications is yet to be explored. This study focuses on examining the timing characteristics of tests implemented as a Metasploit module.

1.2 Contributions

This work contributes to the understanding of performing security tests and assessment of networked embedded systems in the following ways:

- The work systematically analyzes and evaluates the effect of the implementation language, scheduling policy, process priority, JIT Compilation, and use of framework on the performance related to temporal characteristics in Metasploit.
- The work quantitatively analyzes the temporal characteristics of tests or attacks executed using the Metasploit Framework.
- Based on the results, the work provides the best practices and recommendations for using Metasploit to test time-sensitive networked embedded systems.

- The work proposes a framework to integrate microcontroller-based tests with the Metasploit Framework.

1.3 Organization of the Thesis

The rest of the thesis has been organized into six additional chapters. Chapter 2 provides background information on the topics discussed in this thesis. Chapter 3 describes the experiment designed to evaluate the performance of the Metasploit Framework using chosen metrics. Chapter 4 presents the observations made on the results of the experiments. Chapter 5 presents case studies conducted using the Metasploit Framework. Chapter 6 presents the proposed framework to integrate microcontroller-based tests with the Metasploit Framework, and Chapter 7 concludes the thesis.

Chapter 2

Background

2.1 Automotive Cybersecurity

Automotive systems consist of mechanical, electrical and electronic components. Many components in an automotive system are controlled by software. Moreover, components in automotive systems are connected to the Internet. Such inclusion of software and connection to the Internet introduces attack surfaces on automotive systems. For e.g., an automotive system may consist of cameras, Electronic Control Unit (ECU), Light Detection and Ranging (LIDAR), Bluetooth, Wi-Fi, etc. An automotive system may be equipped with Assisted Driving technology, and computer vision algorithms may be running onboard the system. Figure 2.1 shows a typical automotive CAN network [50]. The figure shows two CAN networks, one low-speed CAN bus and another high-speed CAN bus, connected by a CAN gateway. The diagram also shows different automotive ECUs connected to the CAN network. While such features enrich the automotive system, these features also increase additional attack surfaces for an adversary. Hence, cybersecurity should be taken into account for making automotive systems safe.

2.2 Controller Area Network (CAN)

CAN bus has become the de facto standard for communication between Automotive ECUs [30]. CAN is a multi-master, message broadcast network. The CAN communication protocol is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority (CSMA/CD+AMP) [39]. Only one node can transmit a message in

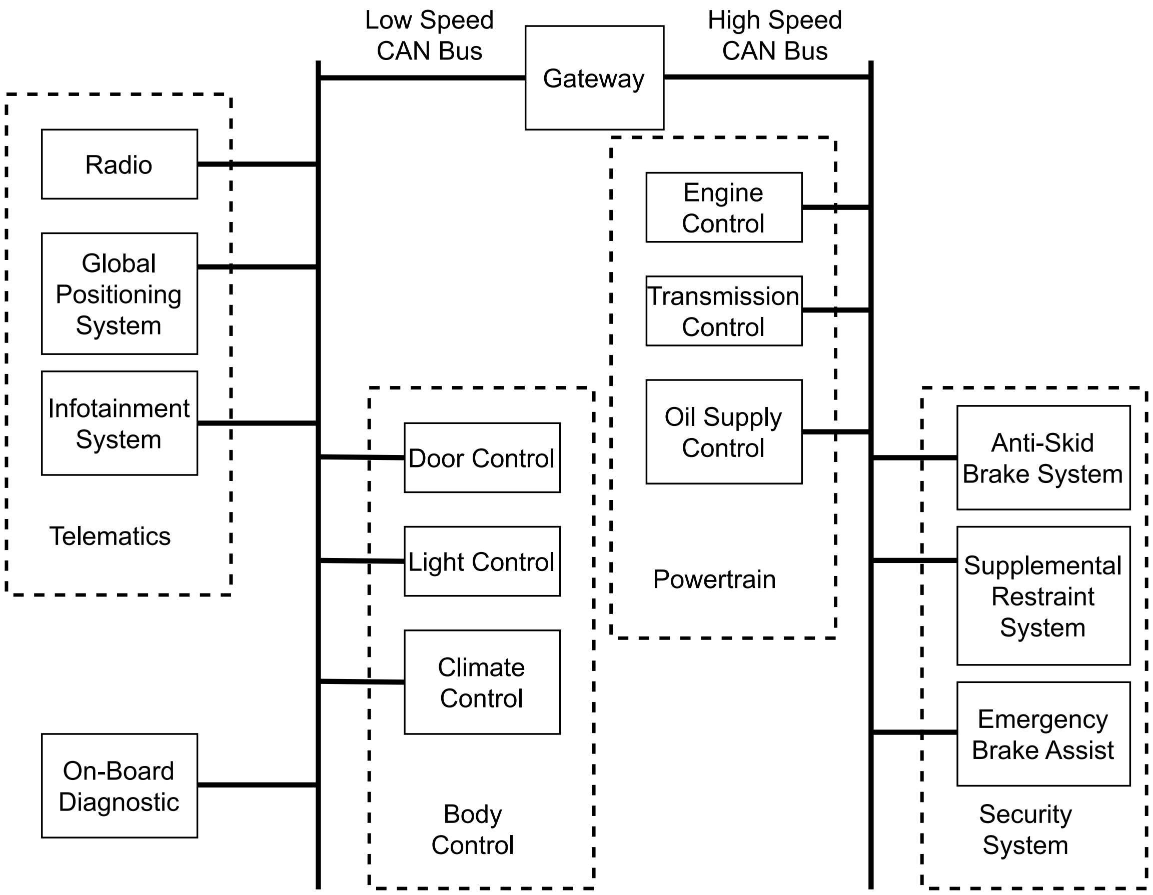

Figure 2.1: Typical automotive CAN network

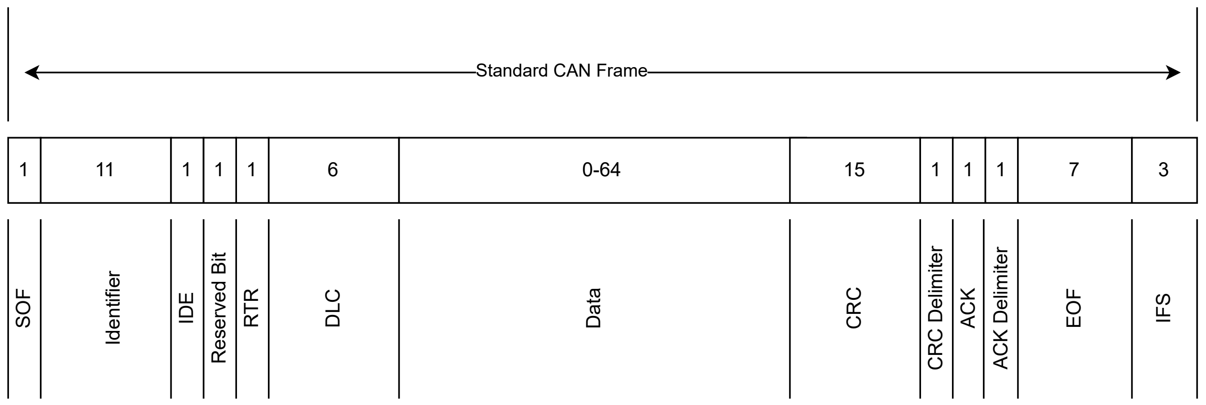

Figure 2.2: Standard CAN frame with field names — the numbers in the boxes represent the number of bits

the network at a particular instant in time. When multiple nodes attempt to transmit a message in the CAN bus simultaneously, bit-wise bus arbitration decides which node gets to transmit its message. The bus arbitration depends on the identifier field of the CAN message frame. A CAN message does not consist of a transmitter or receiver identifier. Instead, each CAN frame consists of a message identifier. The identifier provides the CAN frame its meaning and its priority. The lower the identifier's numeric value, the higher the message priority. A logic-0 in the CAN Network is called a dominant bit, and a logic-1 is called a recessive bit. When two nodes try to write to the bus simultaneously and the bits being written are different in the identifier field, the bus is driven to a dominant bit state. Thus, the node which writes the dominant bit to the bus wins the arbitration.

2.2.1 CAN Frames

There are four types of CAN frames.

Data Frame Data frames consist of actual message or data that the nodes want to convey to other nodes in the network. Figure 2.2 shows the CAN frame format for the standard CAN frame, and Table 2.1 shows the description of each field in the standard CAN frame. Figure 2.3 shows the CAN frame format for the extended CAN frame, and Table 2.2 shows the description of each field in the extended CAN frame.

Field Name	**Length (bits)**	**Description**
SOF	1	Start of Frame
Identifier	11	An identifier to represent the message and message priority
RTR	1	Remote Transmission Request bit. 0 means the frame is a data frame. 1 means the frame is a remote request frame.
IDE	1	Identifier Extension bit. 0 means the frame is a standard CAN frame. 1 means the frame is an extended CAN frame
Reserved Bit	1	Reserved bit. Must be dominant, but it is accepted as either dominant or recessive.
DLC	4	Data Length Code. Denotes the number of bytes in data field.
Data Field	0-64	Data
CRC	15	Cyclic Redundancy Checksum
CRC Delimiter	1	Must be recessive.
ACK	1	Acknowledgement bit. Transmitter sends a recessive bit and receiver can assert a dominant bit.
ACK Delimiter	1	Must be recessive.
EOF	7	End of Frame. Must be recessive.
IFS	3	Inter-Frame Spacing. Must be recessive.

Table 2.1: Description of a standard CAN frame

Field Name	Length (bits)	Description
SOF	1	Start of frame
Identifier A	11	An identifier to represent the message and message priority
SRR	1	Substitute Remote Request. Must be 1.
IDE	1	Remote Transmission Request bit. 0 means the frame is a data frame. 1 means the frame is a remote request frame.
Identifier B	18	Second part of the identifier, which represents the message and message priority.
RTR	1	Remote Transmission Request bit. 0 means the frame is a data frame. 1 means the frame is a remote request frame.
IDE	1	Identifier Extension bit. 0 means the frame is a standard CAN frame. 1 means the frame is an extended CAN frame
Reserved Bit	1	Reserved bit. Must be dominant, but it is accepted as either dominant or recessive.
DLC	4	Data Length Code. Denotes the number of bytes in the data field.
Data Field	0-64	Data
CRC	15	Cyclic Redundancy Checksum
CRC Delimiter	1	Must be recessive.
ACK	1	Acknowledgement bit. Transmitter sends a recessive bit and receiver can assert a dominant bit.
ACK Delimiter	1	Must be recessive.
EOF	7	End of Frame. Must be recessive.
IFS	3	Inter-Frame Spacing. Must be recessive.

Table 2.2: Description of an extended CAN frame

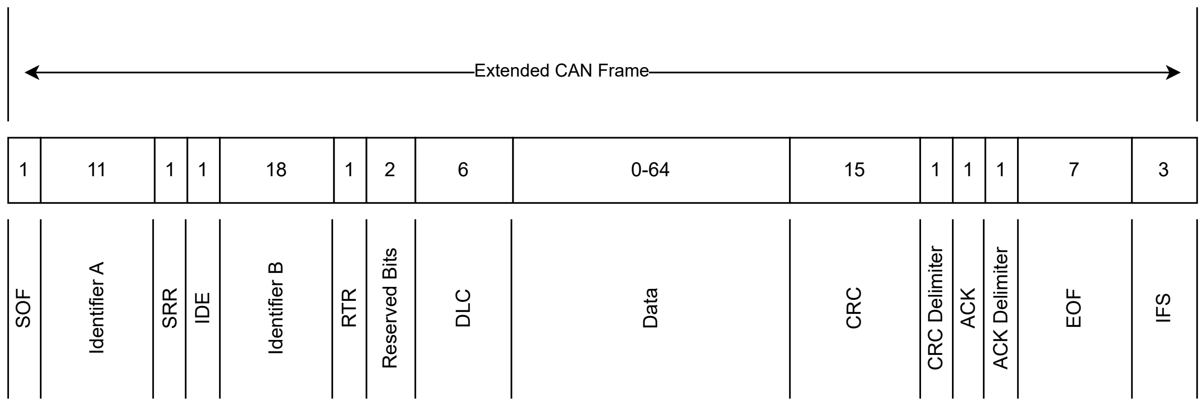

Figure 2.3: Extended CAN frame with field names — the numbers in the boxes represent the number of bits

Remote Frame A node in a CAN network can send a message in the bus autonomously. If a node in the network wants to request data for a specific message identifier, the node can send a remote frame. The Remote Transmission Request (RTR) bit in a remote frame is recessive. A remote frame consists of no data field.

Error Frame Error frame is used to indicate errors in the CAN nodes or in the network using error flags. An error flag can be Active Error Flag (six dominant bits) or a Passive Error Flag (six recessive bits). When Transmission Error Count (TEC) or Reception Error Count (REC) is greater than 127 and less than 255 for a node, the node transmits a Passive Error Frame. When TEC or REC is less than 128, the node transmits an Active Error Frame. The details behind TEC and REC are explained in Section 2.2.2.

Overload Frame A node sends an overload frame to inject additional delay between data or remote frames if the node receives CAN messages faster than the node can process.

2.2.2 Error Handling and Fault Confinement

There are a number of errors in the CAN protocol — Bit Error, Stuff Error, CRC Error, ACK Error, etc. The study concerns itself with a bit error. Each node in the CAN bus monitors whatever it writes onto the bus. If a node writing onto the bus detects that the bus state is not what the node has written, then it detects an error, which is called the

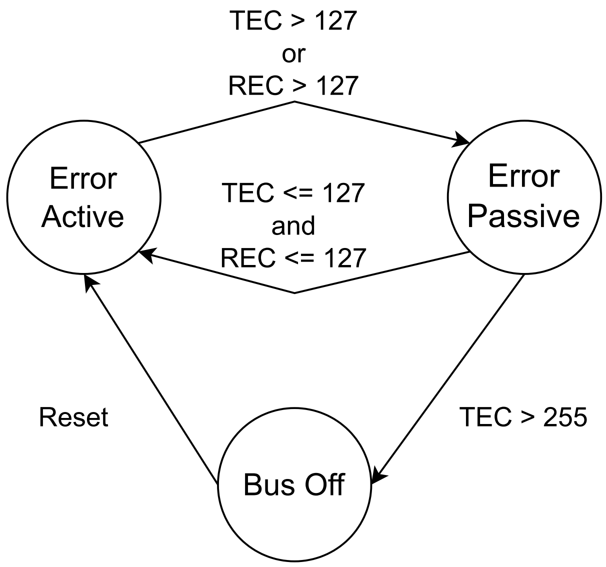

Figure 2.4: State diagram for CAN protocol error handling

bit error. For an error detected by a node, the node transmits an error frame on the bus. Each node maintains two counters: TEC and REC. For an error during transmission, the transmitting node increases TEC by 8. For an error during reception, the receiving node increases REC by 1. For every error-free transmission, TEC is decreased by 1. For every error-free reception, REC is decreased by 1.

Figure 2.4 shows the state transition diagram for error handling in CAN protocol. Each node in the CAN bus starts in an Error-Active state. Depending upon the TEC and REC values, a node may switch to Error-Passive state or Bus-Off state. When TEC or REC exceeds 127, the node goes to Error-Passive state. When TEC exceeds 255, the node goes to the Bus-Off state. In the Bus-Off state, the node loses its ability to transmit and receive CAN messages.

2.3 Security in a CAN Network

The sections below discuss vulnerabilities and attacking methodologies in a CAN network.

2.3.1 Vulnerabilities in CAN Protocol

CAN Protocol was designed to solve the problem of minimizing the wires needed to form a network of ECUs in an automotive system. CAN Protocol was not designed with security in consideration. The CAN Protocol renders itself vulnerable due to following reasons [50]:

Broadcast Communication Any node in a CAN network can receive messages transmitted by any other node in the network. And any node in the network can transmit messages in the network that can be received by any other node in the network.

Lack of encryption The messages flowing in the network can be read and analyzed by any other node in the network.

Lack of authentication A node in a CAN network can impersonate any other node in the network. CAN protocol does not necessitate that a node has an identity in the network. Only messages have an identity in the network. So, any node in the network can transmit messages impersonating the functionality of another node in the network.

Message Identifier-based Priority Scheme The lower the identifier of a message, the higher its priority in the CAN network. This implies a malicious node in the network can deny access to the bus to non-malicious nodes in the network by flooding the network with high-priority messages or messages with low message identifiers.

2.3.2 Attacking Methodologies

This section discusses ways of attacking the nodes in a CAN network.

Frame Sniffing A CAN Network is a broadcast network. So, any node in the network can sniff CAN Frames in the network. By analyzing CAN Frames, functionalities of the ECUs connected to the CAN Network can be known [47].

Frame Injection Since a CAN Network does not have node authentication, once a malicious node gets access to the network, the node can inject frames in the network [50].

Dominant-Bit Assertion CAN's priority-based arbitration scheme allows a node to assert a dominant bit on the bus indefinitely to cause all other nodes in the CAN Network to back off from accessing the network [47]. Palanca et al. [58] used this methodology to mount a Denial-of-Service (DoS) attack.

2.4 Cyber-Attack as a Real-Time Process

This section discusses how an attack on an automotive network can have real-time constraints. Researchers have identified and documented several attacks on the CAN Network [20, 22, 24, 29, 36, 47, 59]. This study demonstrates the role temporal characteristics play in cybersecurity-related tests/attacks using two attacks on the CAN Network (i) CAN-Flood Attack [22, 10] and (ii) Bus-Off Attack [24]. The study chooses the two attacks because the success of the attacks requires certain temporal requirements to be met.

2.4.1 CAN-Flood Attack

The section discusses the threat model and attack model for a CAN-Flood attack.

Thread Model

In a CAN Network, if two nodes try to transmit messages with different CAN IDs simultaneously, the node that sends a message with a smaller CAN ID wins the arbitration. So, if a malicious node frequently sends a CAN message with a low CAN ID, non-malicious nodes do not get the chance to send messages with a higher CAN ID. Such contention for access to the bus leads to the non-malicious nodes experiencing Denial-of-Service.

In a CAN-Flood Attack, the adversary has the capability to transmit and receive messages in the targeted CAN network. The objective of the CAN-Flood attack is to transmit CAN messages with a low CAN ID as frequently as possible.

Attack Model

A CAN Bus network has at least three nodes, including the adversary node. The adversary transmits CAN messages in the network, with message identifier numerically lower than all other CAN messages flowing in the network, as frequently as possible.

Temporal Requirements

The temporal requirement in this attack is to minimize the temporal distance between two CAN messages transmitted by the malicious node.

2.4.2 Bus-Off Attack

The Bus-Off attack [24] exploits the error handling and fault confinement mechanism of CAN communication protocol. The attack focuses on causing a bit-error during the transmission of a message by the target node.

Threat Model

In a CAN network, there's a node V which is the target of the attack. An adversary A has access to the CAN network and can transmit messages to and receive from the network. A's objective is to cause the target node V to enter a *Bus-Off* state such that node V loses its ability to transmit messages to and receive from the network. The adversary A can determine all the information necessary for the attack, such as the CAN ID, message body and the timing of the targeted message.

Attack Model

The objective of Adversary A is to cause a bit-error while Node V is transmitting Message M on the CAN bus. To cause a bit-error, Adversary A needs to start sending the message at exactly the same instant as Node V with the same message identifier that is being sent by Node V. In such a case, both the nodes A and V win the bus arbitration, and both nodes continue sending the rest of the message. To cause the bit-error, Node A needs to craft the message body such that in one of the bit positions, Message M is recessive, whereas Message C is dominant. When Node V sends a recessive bit and Adversary A writes a dominant bit, the bus goes to a dominant bit state. Node V detects an error and increments its TEC by 8. By repeatedly causing a bit error in the message transmitted by Node V, Node V's TEC eventually exceeds 255. Consequently, Node V enters the *Bus-Off* state. To predict the instant at which Node V starts writing the message onto the bus, Adversary A targets a message M, which is sent by Node V periodically.

Attack Setup

There are three nodes in a CAN network: $\mathcal{X}$, $\mathcal{V}$ and $\mathcal{A}$, where $\mathcal{X}$ is the trigger node, $\mathcal{V}$ is the target node, and $\mathcal{A}$ is the adversary node. The Bus-Off attack is executed with the help of preceded messages. The attack events progress as follows:

- $\mathcal{X}$ transmits Message $\mathcal{K}$ periodically with a period T.
- Immediately after transmitting Message $\mathcal{K}$, $\mathcal{X}$ transmits k preceded messages P_1, P_2, ..., P_k.
- When $\mathcal{V}$ and $\mathcal{A}$ receive Message $\mathcal{K}$, $\mathcal{V}$ transmits Message $\mathcal{M}$ (targeted message) and $\mathcal{A}$ transmits Message $\mathcal{C}$ (attack message). The attack and target messages must have the same CAN ID. The attack message must differ from the target message such that a bit in the attack message is dominant, whereas the corresponding bit in the target message is recessive.

Temporal Requirements

If the instant at which $\mathcal{V}$ and $\mathcal{A}$ start writing the messages $\mathcal{C}$ and $\mathcal{M}$ onto the bus differs by even a single bit, the attack fails. Considering a bitrate of 500 kbps, the bit-length is 2μs. By using the preceded messages, the timing requirement of the attack message injection becomes relaxed. Instead of 1 bit-length, the timing requirement is now the message length of the preceded message.

The study takes into account the difference in CAN message round-trip latency of the target and the adversary nodes. Let the delay between trigger message $\mathcal{K}$ and target message $\mathcal{M}$ be D_{target}, in the absence of preceded messages. Let the delay between $\mathcal{K}$ and attack message $\mathcal{C}$ be D_{attack}, in the absence of preceded messages. Due to jitter, the delays or round-trip times for the nodes are not deterministic [40, 24]. Let $D_{target,max}$ be the upper bound on D_{target} and $D_{attack,max}$ be the upper bound on D_{target}.

Let the temporal lengths of trigger, precede, target and attack messages be $t_{trigger}$, $t_{precede}$, t_{target} and t_{attack}, respectively. If t_k is the instant of the start of the trigger message, then the preceded messages should keep the bus busy until at least
$t_k + max(D_{target,max}, D_{attack,max}) - t_{trigger}$, i.e.,

$$k * t_{precede} \geq (max(D_{target,max}, D_{attack,max}) - t_{trigger}) \tag{2.1}$$

where, k is the number of preceded messages required to keep the bus busy such that the Messages $\mathcal{M}$ and $\mathcal{C}$ synchronize.

2.5 The Metasploit Framework

The Metasploit Framework is a platform for penetration testing built using the Ruby programming language. It consists of a suite of tools to test security vulnerabilities, enumerate networks, execute attacks, and evade detection. The Metasploit Framework is focused on network and software penetration testing. The Hardware Bridge API broadens the scope of the Metasploit Framework by expanding its capabilities onto the physical world of hardware devices. The Hardware Bridge API aims to unify tools pertaining to security testing with the use of hardware devices. Some of the devices that have been used with the help of the Hardware Bridge API are CAN, ZigBee, Bluetooth, OBD-II, etc.

2.6 Hypothesis Testing

Given any one or two sample distributions, hypothesis testing can be used to decide if the given distributions support a particular hypothesis or not. The decision is made based on a probabilistic inference on the given sample distribution(s). A hypothesis is stated in terms of a null hypothesis (H_0) and an alternative hypothesis (H_1). There can be different types of hypothesis testing which are discussed below.

2.6.1 Nature of Sample(s)

Based on the nature of the sample(s), a hypothesis test can be categorized into the following three categories.

One Sample Test

If the test compares a sample distribution against the population distribution, then the test is called a One Sample Test.

Independent Samples Test

In an Independent Samples Test, the idea is to compare one sample distribution with another sample distribution, not a population distribution. The two samples are independent as opposed to paired, which is discussed below.

Paired Samples Test

In a Paired Samples Test, the test compares one sample distribution X with another sample distribution Y, where Y consists of the same subjects as X. The difference between X and Y is that the subjects of X have gone through treatment or some process about which the hypothesis is getting tested.

2.6.2 Direction of Test

The null hypothesis states that there is no "significant" difference between certain characteristics of the two distributions being compared. If the characteristic of interest is the mean of the sample distribution, then the null hypothesis states that the two means are the same. Based on the alternative hypothesis, the test can be categorized into the following categories.

Two-tailed Hypothesis Testing

Let us consider the mean of the distributions μ_1 and μ_2. In a two-tailed hypothesis testing, the null hypothesis states that u_1 equals μ_2. The alternative hypothesis is that μ_1 is either greater than or less than μ_2.

$\mathbf{H_0} : \mu_1 = \mu_2$

$\mathbf{H_1} : \mu_1 \neq \mu_2$

One-tailed Hypothesis Testing

In one-tailed hypothesis testing, the alternative hypothesis states greater than or less than the relation between the two parameters. If the alternative hypothesis states a greater than relationship, then the hypothesis can be given as:

$\mathbf{H_0} : \mu_1 \leq \mu_2$

$\mathbf{H_1} : \mu_1 > \mu_2$

If the alternative hypothesis states a less-than relationship, then the hypothesis can be given as

$\mathbf{H_0} : \mu_1 \geq \mu_2$

$\mathbf{H_1} : \mu_1 < \mu_2$

2.6.3 Tests for Comparing One or Two Sample Distributions

Hypothesis testing may be one-tailed or two-tailed, and the testing may be one-sample or independent-samples or paired-samples. There can be a number of tests for hypothesis testing. Once a test is chosen and the test statistic is calculated, we check if the calculated test statistic falls in the critical region or not. If the test statistic falls in the critical region, we reject the null hypothesis and accept the alternative hypothesis. The critical region is decided using a significance level (α) and the corresponding confidence interval. For e.g. a significance level of 0.05 means that we are willing to accept a 5% error in rejecting the null hypothesis or we have a confidence interval of 95% in rejecting the null hypothesis. The tests calculate the p-value, which is the probability that favours the null hypothesis. If the p-value lies in the critical region, then we reject the null hypothesis.

Based on the nature of distribution, i.e., known population variance, homogeneity of variance, and normality of the distribution, one of the following tests can be chosen. The z-test makes the following assumptions:

1. Population distribution is known.
2. Population variance is the same as the sample variance.
3. The sample distribution follows a normal distribution.

If the assumption about known population distribution cannot be met, Student's t-test [70] can be used. The Student's t-test makes the following assumptions:

1. Sample distribution follows a normal distribution.
2. The two distributions being compared have the same variance.

When comparing two sample distributions, if the assumption about the variance of two distributions being the same cannot be met, Welch's t-test [71] can be used. Welch's t-test makes the assumption that the sample distribution follows a normal distribution.

If the data does not follow a normal distribution, then non-parametric tests like Wilcoxon Rank Sum Test [73] can be used for independent samples test. For paired samples, Wilcoxon Signed Rank Sum Test can be used. Wilcoxon Rank Sum Test tests the hypothesis that two randomly selected samples from distributions, say X and Y, are different. However, the test still assumes equal variances in the sample distributions. Fligner and Policello [34] suggested the Robust Rank Order Test for sample distributions which are not normal distributions and do not have equal variances [44]. The Fligner-Policello [34] test assumes a continuous distribution, i.e. no ties. The Brunner-Munzel Test [21] extends the test to allow for unequal variance and discrete or ordered categorical random variables.

2.6.4 Tests for Comparing More Than Two Sample Distributions

The tests discussed in Section 2.6.3 can be used for one sample, two independent samples or two paired samples. One-Way Analysis of Variance (ANOVA) [32, 33] can be used when comparing more than two distributions on a single independent variable when the distributions follow the assumptions stated for Student's t-test. For n independent samples, the hypothesis can be stated as

$\mathbf{H_0}$ **:** $\mu_i = \mu_j; \forall i, j = 1, ..., n$

$\mathbf{H_1}$ **:** $\exists i, j : 1, ..., n \textit{ such that } \mu_i \neq \mu_j; i \neq j$

The null hypothesis says that all the distributions have the same statistic (say mean). The alternative hypothesis says that one or more statistic is different for the sample distributions.

One-Way ANOVA cannot be used if the distributions violate the assumptions made for Student's t-test. One-Way Welch's ANOVA [72] can be used when comparing more than two distributions on a single independent variable when the distributions follow the assumptions stated for Welch's t-test.

Kruskal-Wallis Test [48] is a non-parametric equivalent of ANOVA for multiple sample distributions. Kruskal-Wallis Test can be used for non-normal sample distributions with equal variances.

2.6.5 p-value Correction

When performing any hypothesis testing, there's a certain error associated with rejecting the null hypothesis, even when the null hypothesis is true for the population. This error is called Type-I error (α) or False Positive Rate. When performing multiple tests, we need to consider the Family-Wise Error Rate (FWER), which is the probability of seeing at least one significant result erroneously. When performing m multiple tests,

$$FWER = 1 - (1 - \alpha)^m \tag{2.2}$$

So to maintain FWER less than the desired value of α, calculated values of the p-value for each test must be adjusted. Various methods exist for p-value corrections - Bonferroni's Correction [28], Holm's Method [38], etc.

2.6.6 Effect Size

The tests such as the ones discussed in Section 2.6.3 and Section 2.6.4 allow making a decision on whether an alternative hypothesis should be accepted or not. However, the tests do not provide any information on if any two distributions are different or how much they differ. It is important to specify the effect size in such a case. Effect size provides a way to check the magnitude of the difference between the two distributions. For normally distributed data, when Student's t-test or Welch's t-test is used for hypothesis testing, effect size such as Cohen's d [25] can be used. When hypothesis testing is done using Wilcoxon Rank Sum Test or Brunner-Munzel Test, effect size such as Common Language Effect Size (CLES) [52] can be used. CLES states the probability that a sample from one distribution is smaller than a sample from another distribution.

Chapter 3

Experiment Design

The study aims to evaluate and improve the temporal characteristics of modules in Metasploit. There are a number of choices available, from the creation of the software to the execution of the software. The study considers such choices as *factors* that affect the temporal characteristics of a task. The study considers the following factors: 1) Programming Language, 2) Process Scheduling Policy and Process Priority 3) Periodicity of a Periodic Task 4) JIT Compilation 5) Effect of using the Metasploit Framework 6) Hardware Setup. To determine which combination of factors provides better temporal control, we perform experiments to estimate 1) jitter in the periodicity (Δt_p) of CAN message transmissions 2) transmission latency and jitter (Δt_{tx}) for a CAN message 3) round-trip latency and jitter (Δt_{rt}) for a CAN message under different combinations of factors. Additionally, jitter in the periodicity of periodic CAN message transmissions is estimated for different periods along with the program configurations. The experiments use standard CAN frames.

3.1 Factors affecting Temporal Characteristics

There are a number of choices available from the creation of software to the execution of the software. The study considers such choices as the “factors” in the timing of a task. There are a number of factors that can affect the timing of a process executing in an operating system.

3.1.1 Programming Language

The language used to write a program is very significant with respect to the speed of execution and execution jitter. Metasploit is implemented using the Ruby programming language. Metasploit modules are written in Ruby as well. Ruby is not a systems programming language and is not efficient or fast compared to a systems programming language like C [57]. C is a compiled language whereas Ruby is an interpreted language. The study performed by Heer shows that a (CPU intensive) program written in C is nearly 188 times faster than a corresponding program written in Ruby [14]. The task chosen in this study is I/O intensive as opposed to CPU intensive. Nonetheless, the authors expect the C program to be faster than the corresponding Ruby program.

3.1.2 Scheduling Policy and Process Priority

A General Purpose Operating System like Ubuntu (Linux) has several processes running at once. When several processes are vying for CPU time, the scheduling policy and process priority can affect the timing of the processes. The exposition in [46] provides information on the different process priorities and scheduling methods used in Linux. SCHED_OTHER is the default scheduling method for processes in Linux. Processes with SCHED_OTHER scheduling get priorities in the range of 100-139, which are provided in the form of nice values. The priorities 100-139 are mapped to nice values -20 to +19, -20 being the highest priority. The default nice value of a process with SCHED_OTHER scheduling is 0. Linux also provides SCHED_FIFO and SCHED_RR for processes that need real-time scheduling. The processes with SCHED_FIFO or SCHED_RR scheduling get the priorities in the range of 0-99, 99 being the highest. The study of the effect of the scheduling policies under different task compositions is out of the scope of the study. The study considers default task composition which is in the system under fresh install of the Ubuntu 20.04 Linux distribution, plus essential tasks like a code editor, a terminal to execute tests and the tests themselves.

3.1.3 Period of a Periodic Task

For evaluation of jitter in periodic CAN messages, the experiment uses the sleep method (actual sleep function used is shown in Table 3.2). Calling the sleep method in a program causes the program to suspend the execution for a specified amount of time. One of the questions this study considers is: does jitter in periodicity depend upon the duration of sleep? To answer this question, this study conducts experiments with different sleep durations.

3.1.4 Effect of using Metasploit

Metasploit is a framework and specific tests in this framework are implemented as modules. The framework provides access to hardware in the form of APIs, which may affect the timing characteristics of the operation under consideration. For the task under consideration i.e., transmitting and receiving CAN messages, Metasploit provides APIs [12] which are wrappers around *can-utils* [11] (userspace utilities for SocketCAN [4]). This study compares how the timing characteristics of an operation implemented as a Metasploit module differs from a standalone implementation.

3.1.5 Just-in-Time (JIT) Compilation

A program runs faster when the program is in a form directly executable on the processor [17]. JIT compilation is one of the approaches taken to speed up the execution of programs written in interpreted languages. In JIT compilation, the programs are compiled during the first run such that the subsequent runs of the program or parts of the program are faster. Ruby is an interpreted language and provides the option to run Ruby programs with JIT enabled. To determine the effect of JIT compilation on the temporal characteristics of the tests, the experiments conducted include standalone Ruby programs with and without JIT enabled.

3.1.6 Hardware Setup

The hardware setup, for e.g., the CPU, and the I/O devices, used can affect the timing of the task. To communicate with the CAN network, a general-purpose computer needs to use a USB-to-CAN module. The specific module used adds latency and jitter to the communication. Pozzobon et al. conducted a survey on the latency introduced by different USB-to-CAN modules [61]. The study does not cover all the USB-to-CAN modules available. A detailed study on the effect of the hardware setup used is not in the scope of this study. However, the study does consider a microcontroller platform and proposes a framework to integrate tests on a microcontroller with Metasploit in Chapter 6. For the proposed framework, the study chooses Arduino Uno [16] as the choice of the microcontroller.

3.2 Program Configuration

This study uses the term *configuration* to denote a particular choice of *factor*. Instead of defining a separate configuration for each factor, we group some of the factors. Table 3.1 shows the different configurations and the values each configuration can have.

Framework Configuration Framework configuration tells if the program is a standalone program or, implemented as a Metasploit module or running in a microcontroller such as Arduino Uno. A framework configuration can be *Standalone* or, *Metasploit* or, *Arduino*.

Language Configuration Language configuration shows which language is used to implement the program. The values for this configuration can be *C*, *Ruby* and *Ruby:JIT*. Here, *Ruby* means the program written in Ruby is executed without enabling the JIT feature of the Ruby interpreter. *Ruby:JIT* means the program written in Ruby is executed with JIT enabled. Additionally, the study introduces another language configuration *Ruby:C*. *Ruby:C* means that the functionality is implemented as a function in a C shared library, which is then called from a Ruby program using Foreign Function Interface (FFI) [53]. This particular configuration is important because Metasploit modules are implemented in Ruby. So, to utilize the advantages offered by C, the actual intended functionality of the module is implemented in C as a shared library function which can then be called from a Metasploit module written in Ruby.

Scheduling Configuration Scheduling configuration combines the scheduling policy and process priority factors. The study considers three scheduling configurations: (i) *Default*, (ii) *FIFO*, and (iii) *RR*. *Default* scheduling configuration means that the program, when executed, uses the default scheduling policy, i.e. SCHED_OTHER and the default nice value of 0. *FIFO* scheduling configuration means that the scheduling policy used for the process is SCHED_FIFO and the process priority used is 99. *RR* scheduling configuration means that the scheduling policy used for the process is SCHED_RR and the process priority used is 99.

Naming Convention This study refers to a combination of individual configurations as a *program configuration*. To name the different program configurations used in the experiments, this study uses the naming convention of *[Framework Configuration] - [Language*

Configuration	**Values**
Framework	Metasploit, Standalone, Arduino
Language	C, Ruby, Ruby:JIT, Ruby:C
Scheduling	Default, FIFO, RR

Table 3.1: Individual configurations and their possible values

Configuration] - [Scheduling Configuration]. For example, a program configuration mentioned as *Standalone-C-FIFO* means (i) the program is a standalone program (i.e., not a Metasploit module, or an Arduino Uno program), (ii) the program is implemented in C, and (iii) the program is executed with SCHED_FIFO scheduling policy and a process priority of 99. Table 3.2 shows all the configurations that the study considers along with the sleep function used to achieve periodicity in Experiment 1 and the tool/library/API used to interact with CAN bus. To imply that a particular discussion is applicable to all the values of a specific configuration, the thesis uses the wildcard character - asterisk ($*$). For example, *Metasploit-Ruby-*$*$ means that the discussion is applicable for the program type configuration of Metasploit, language configuration of Ruby and all scheduling configurations. Since the study uses the Arduino Programming Language [1] for Arduino Uno and scheduling configuration is not applicable to Arduino Uno, the program configuration that uses Arduino Uno is shortened to *Arduino*.

3.3 Experimental Setup

The experimental setup is shown in Figure 3.1. The experiment assumes a test/attack scenario in which the system-under-test is able to transmit and receive CAN messages. The figure shows a CAN bus connection between two CAN nodes. Node 1 is the system-under-test (SUT). For *Metasploit* and *Standalone* framework configurations, the SUT node is a 4-core i5 computer with 16 GB of RAM, and each core has a frequency of 1.60 GHz. The SUT node uses the Metasploit framework - version *6.0.55-dev* running on Ubuntu 20.04 Linux Distribution. The average CPU load when no tests are running is nearly 0.05%. The USB-to-CAN module used in Node 1 is *Peak CAN* [60]. For *Arduino* framework configuration, the SUT node is an Arduino Uno with a MCP2515 [43] CAN controller and MCP2551 [42] CAN transceiver. The nodes are configured to use a bitrate of 500 kbps. A CAN network requires at least two nodes in the network, and Node 2 completes the requirement along with Node 1, which is the SUT. The setup consists of Salae Logic

S. No.	Program Configuration	Sleep Function	CAN Interface
1	Metasploit-Ruby-Default	sleep [9]	APIs provided by Metasploit [12]
2	Metasploit-Ruby-FIFO		
3	Metasploit-Ruby-RR		
4	Metasploit-Ruby:C-Default	usleep [5]	SocketCAN [4]
5	Metasploit-Ruby:C-FIFO		
6	Metasploit-Ruby:C-RR		
7	Standalone-Ruby-Default	sleep [3]	can-utils [11]
8	Standalone-Ruby-FIFO		
9	Standalone-Ruby-RR		
10	Standalone-Ruby:JIT-Default		
11	Standalone-Ruby:JIT-FIFO		
12	Standalone-Ruby:JIT-RR		
13	Standalone-Ruby:C-Default	usleep [5]	SocketCAN [4]
14	Standalone-Ruby:C-FIFO		
15	Standalone-Ruby:C-RR		
16	Standalone-C-Default		
17	Standalone-C-FIFO		
18	Standalone-C-RR		
19	Arduino	delay [2]	arduino-CAN library [55]

Table 3.2: List of program configurations along with sleep function and CAN interface used for the configurations

Analyzer to record the CAN messages along with timestamps for analysis. The logic analyzer is configured to sample the signals at 16MHz.

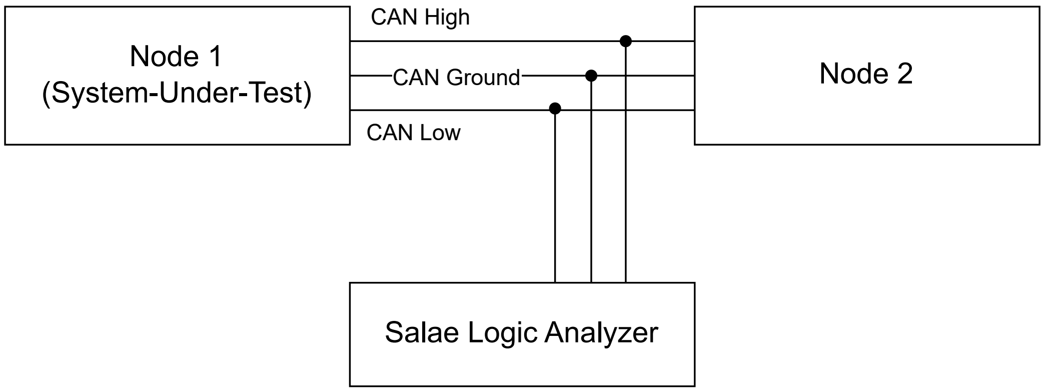

Figure 3.1: Experimental setup showing a CAN network with two nodes and Logic Analyzer connected to the CAN bus.

3.4 Experiments

The study performs the following three experiments to evaluate temporal characteristics for different program configurations.

3.4.1 Experiment 1: Jitter in Periodicity

Experiment 1 is designed to estimate the jitter in periodicity. Node 1 executes the task of sending a CAN message periodically with a period $T_{desired}$. If t_i is the instant at which i_{th} message is seen on the CAN bus, then the jitter for i_{th} transmission ($\Delta t_{p,i}$) is calculated as,

$$\Delta t_{p,i} = t_i - t_{i-1} - T_{desired} \tag{3.1}$$

Let the mean jitter in sleep duration be μ_p and the standard deviation be σ_p. The periodic task is realized by Pseudocode 1. The experiment is performed for 1000 iterations for each program configuration for the sleep durations { 0.1s, 0.5s, 1.0s, 2.0s, 5.0s, 10.s, 20.0s }.

Pseudocode 1: Periodic CAN Message Sender

```
Input: Sleep Duration $T$, No. of iterations $N$
for $i$=0; $i$<=$N$; $i$++ do
    send_can_msg();
    call_sleep_function(T);
```

3.4.2 Experiment 2: Transmission Latency

Experiment 2 is designed to estimate latency and jitter in the transmission of a CAN message. The transmission latency is the time required between the initiation of a transmission of a message from the program/application until the actual presence of the message on the bus. Pseudocode 2 realizes the task to be performed for Experiment 2. Node 1 sends two CAN messages without any explicit delay between the two transmissions. Then the transmission latency (Δt_{tx}) is the temporal spacing between the two messages. Let μ_{tx} be the mean transmission latency and σ_{tx} be the standard deviation in the transmission latency called jitter in transmission latency.

Pseudocode 2: Program under test for the estimation of Δt_{tx}

```
Input: Sleep Duration $T$, No. of iterations $N$
for $i$=0; $i$<$N$; $i$++ do
    send_CAN_message_1();
    send_CAN_message_2();
    sleep_for_some_specific_time($T$);
```

Let $t_{0,i}$ be the instance of time at the start of i_{th} CAN-Message-1. Let $t_{1,i}$ be the instance of time at the start of i_{th} CAN-Message-2. Let $t_{0,length}$ be the temporal length of CAN-Message-1. Then, the temporal spacing between i_{th} pair of CAN-Message-1 and CAN-Message-2 is given as,

$$\Delta t_{tx,i} = t_{1,i} - t_{0,i} - t_{0,length} \tag{3.2}$$

The temporal spacing between the two messages is calculated using Saleae Logic Analyzer 16. The experiment is performed for 1000 iterations for each configuration.

3.4.3 Experiment 3: Round-Trip Latency

Experiment 3 is designed to estimate the round-trip latency of the system-under-test in a CAN network. The round-trip latency is defined as the amount of time it takes for a CAN message to reach the application and for the CAN message transmitted by the application as the response to appear in the CAN bus. Pseudocode 3 realizes the task to be performed for this experiment. Node 2 sends a CAN message (say $P_{e,0}$) on the network. On receiving Message $P_{e,0}$, Node 1 sends another message (say $P_{e,1}$) on the network. Let the round-trip latency (Δt_{rt}) be defined as the temporal spacing between the two messages $P_{e,0}$ and $P_{e,1}$. Let the mean round-trip latency be μ_{rt} and the jitter in round-trip latency be the standard deviation σ_{rt}.

Let $t_{0,i}$ be the instance of time at the start of the trigger message i. Let $t_{1,i}$ be the instance of time at the start of the triggered message. Let $t_{0,length}$ be the temporal length of the trigger message. Then, the temporal spacing between i_{th} pair of trigger messages and the triggered message is given as,

$$\Delta t_{rt,i} = t_{1,i} - t_{0,i} - t_{0,length} \tag{3.3}$$

The temporal spacing between the two messages is calculated using Saleae Logic Analyzer 16. The experiment is performed for 1000 iterations for each program configuration.

Pseudocode 3: Program under test for the estimation of Δt_{rt}

```
  Input: No. of iterations N
1 for i=0; i<N; i++ do
2     wait_for_CAN_message();
3     send_CAN_message();
```

Chapter 4

Observations

This chapter discusses the results of the experiments described in Section 3.4. Figure 4.1 shows the plot of mean jitter in the periodicity of transmitted CAN messages for different sleep durations and different program configurations, as observed in Experiment 1. *Standalone-Ruby-$*$* and *Standalone-Ruby:JIT-$*$* program configurations are indicated using dotted lines and 'X' markers. *Metasploit-Ruby-$*$* program configurations are indicated using solid lines with circle markers. Program configurations which use a C-based implementation (C or Ruby:C language configurations) do not have any markers. The markers hide the error bars in some of the plot lines.

Figure 4.3 shows the box plot for transmission latency for different program configurations, as observed in Experiment 2. The data summary is given in Table A.3. Figure 4.4 shows the box plot for round-trip latency for CAN messages for different program configurations, as observed in Experiment 3. The data summary for Experiment 3 is given in Table A.5.

The work compares data between different program configurations to draw conclusions. From Shapiro-Wilk test [66], we find that the sample distributions do not follow a normal distribution. From Levene's test [49], we find that the variances of the sample distributions are not equal. Since the sample distributions are non-normal, we use non-parametric tests for hypothesis testing. The sample distributions do not have the same variance. Hence, from the non-parametric test, we use Brunner-Munzel [21] test for hypothesis testing. The study uses a significance level of 0.05 for drawing conclusions based on the p-values. To evaluate the effects of framework, language and scheduling configurations, Section 4.1, Section 4.2 and Section 4.3 use the transmission latency data (shown in Figure 4.3 and Table A.3), round-trip latency data (shown in Figure 4.4 and Table A.5) and periodicity

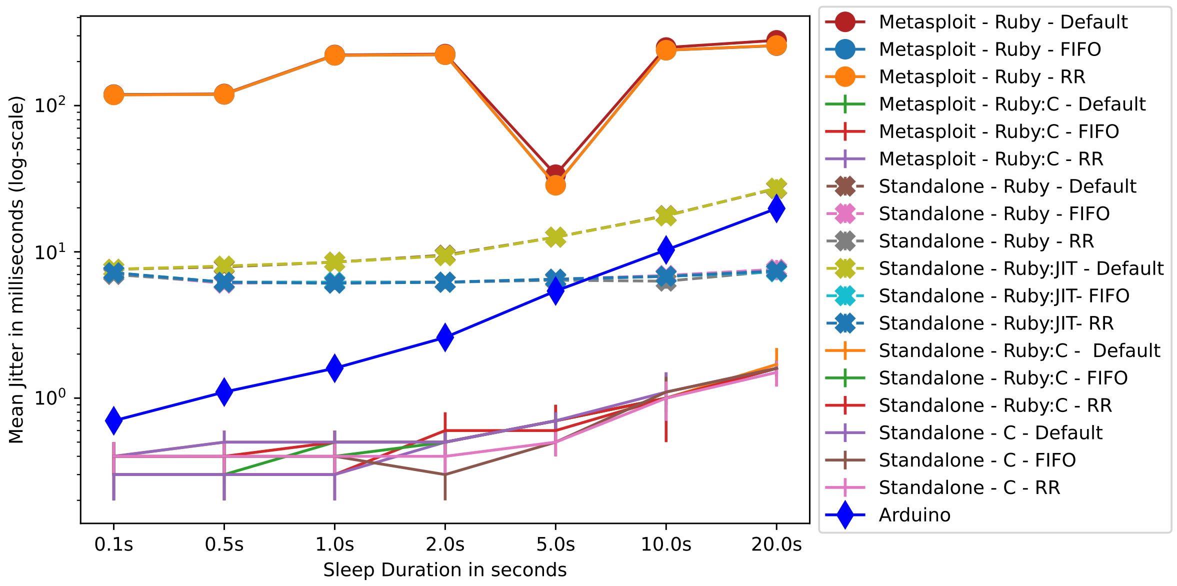

Figure 4.1: Plot of mean jitter in periodic CAN transmission versus sleep duration for different program configurations

jitter data for 5.0s (shown in Figure 4.2 and Table A.1). The study makes the choice to use periodicity jitter data for only a single value of periodicity in order to simplify the analysis. The choice of using 5.0s as the value of periodicity is discussed in Section 4.4. The analysis already involves three independent variables (framework, language and scheduling configurations) for three experiments. The work discusses the effect of periodicity in Section 4.4.

4.1 Effect of using Metasploit Framework

To evaluate the effect of using the Metasploit Framework instead of a standalone program for a test, we compare *Metasploit-Ruby-∗* program configuration with *Standalone-Ruby-∗*. Rows 22, 23 and 24 in Table A.2, Table A.4, and Table A.6 show these comparisons. In all three experiments, the *Standalone-Ruby-∗* program configurations have lower latency and jitter compared to *Metasploit-Ruby-∗*.

The APIs provided by the Metasploit Framework seem to be adding overhead in accessing the hardware interface. The *Standalone-Ruby-∗* program configurations use the

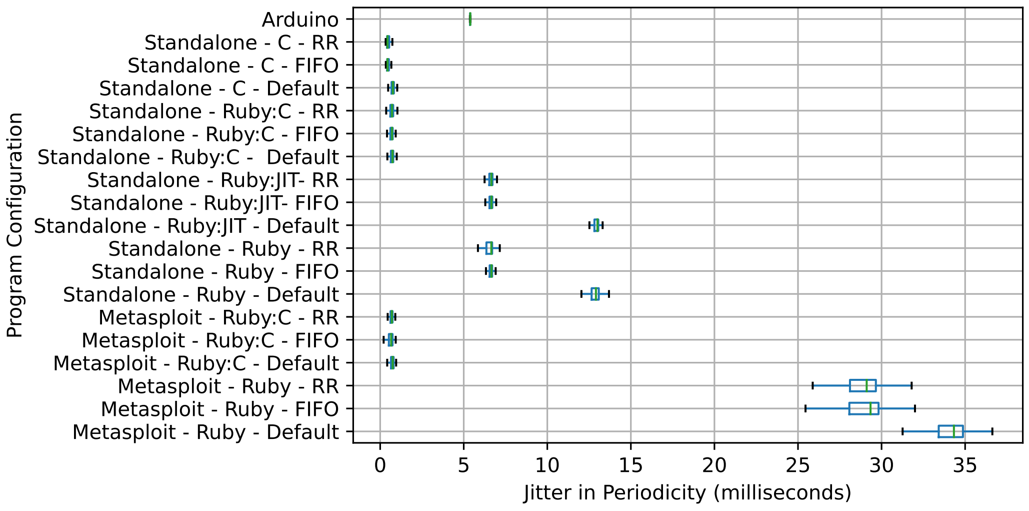

Figure 4.2: Box-Plot of jitter in periodic CAN message transmission with a period of 5.0s

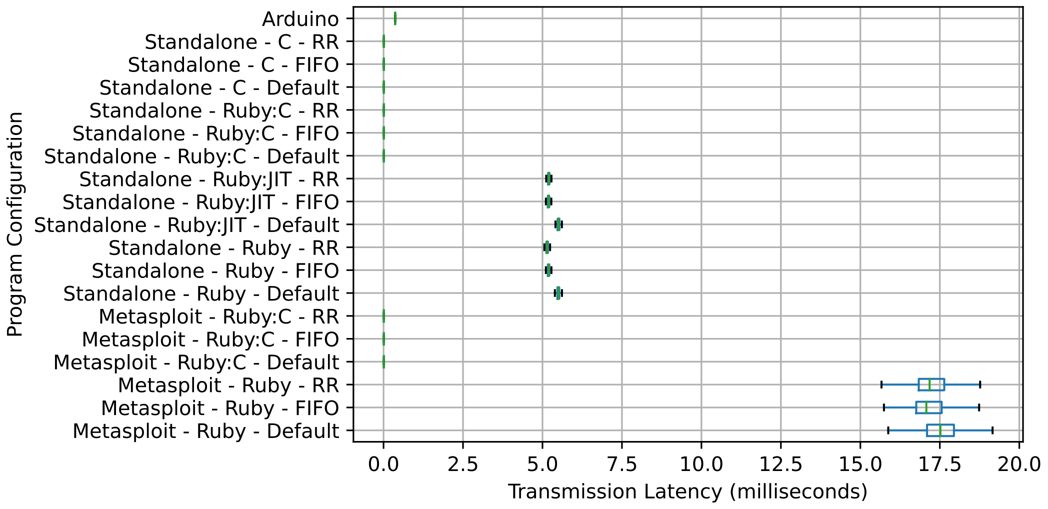

Figure 4.3: Box-Plot of transmission latency for different program configurations

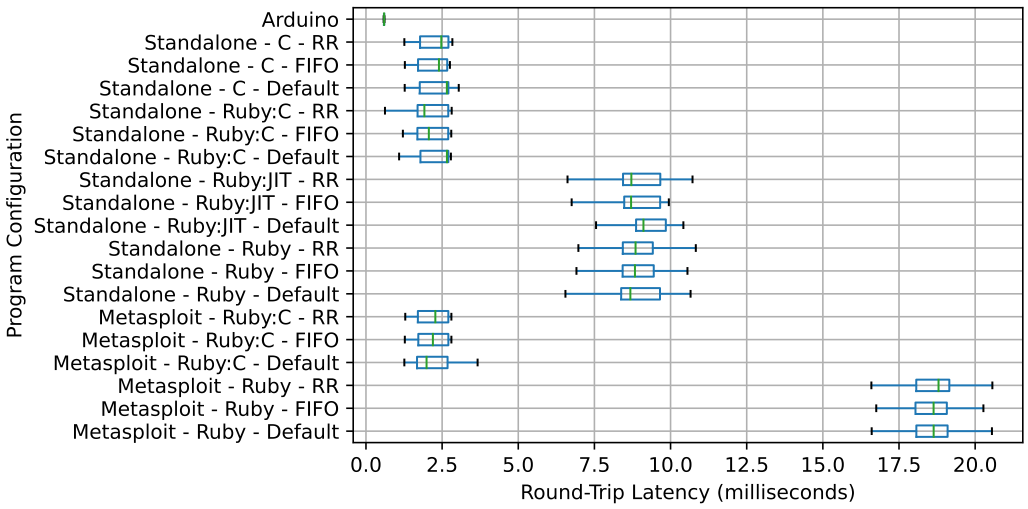

Figure 4.4: Box-Plot of round-trip latency for different program configurations

commandline utilities *cansend* and *candump* to transmit and receive CAN messages. The *Metasploit-Ruby-$*$* configuration uses the same commandline utilities with a wrapper over the utilities.

With respect to the sleep function in *Metasploit-Ruby-$*$*, for the sleep durations considered, the actual sleep was realized by the program configuration using calls to *ppoll* system call [7]. Each call to *ppoll* is made for a duration of 200ms. For a sleep duration of 5.0s, 25 calls to *ppoll* are made with a duration of 200ms in each call. For *Standalone-Ruby-$*$* program configuration, a single call to *ppoll* is made with the actual duration of the sleep.

4.2 Effect of Language Configurations

Language configuration had the most pronounced effect on the temporal characteristics in the experiments. The section first discusses the effect of JIT compilation in the case of standalone Ruby programs. The section compares standalone Ruby programs with and without JIT compilation. Rows 28, 29 and 30 in Table A.2, Table A.4, and Table A.6 show that there's no evidence that JIT compilation in *Standalone-Ruby:JIT-$*$* programs improve the temporal characteristics of the tests compared to *Standalone-Ruby-$*$*.

For all the experiments performed, program configurations using *C* or *Ruby:C* language configurations had the least latency and jitter. The study compares the *Metasploit-Ruby:C-*

Default configuration with all the other configurations which use *Ruby* language configuration. Rows 1 through 9 in Table A.2, Table A.4 and Table A.6 show the comparison. The comparison shows that, in each of the three experiments, *Metasploit-Ruby:C-Default* has lower latency and jitter (i.e., p-value < 0.05) compared to *Metasploit-Ruby-*$*$. Using *strace* to monitor the system calls *Metasploit-Ruby-*$*$ is making in order to transmit a CAN message, we found that *Metasploit-Ruby-*$*$ is making more system calls compared to *Metasploit-Ruby:C-*$*$, as shown in Table 4.1.

One of the differences between using Ruby programming language and using C programming language is the number of calls made to open a connection to CAN socket and close it. The programs written using Ruby programming language use *can-utils* to transmit or receive CAN messages. Each call to *can-utils* utility opens and closes the connection. For e.g. calling *cansend* 1000 times to transmit 1000 messages results in 1000 opening of connection to CAN socket followed by 1000 closing of the connection. The programs written using C programming language use the SocketCAN library directly, which provides the flexibility to open the connection to CAN socket once, transmit 1000 messages and then close the connection.

Comparing *Standalone-C-*$*$ with *Metasploit-Ruby:C-*$*$ (rows 25, 26 and 27 in Table A.2, Table A.4 and Table A.6), the study finds that *Metasploit-Ruby:C-*$*$ is as good as *Standalone-C-*$*$ in terms of temporal characteristics. In most of the Brunner-Munzel [21] tests, the p-values are greater than the significance level, indicating that *Standalone-C-*$*$ does not have better temporal characteristics compared to *Metasploit-Ruby:C-*$*$. In the Brunner-Munzel tests where p-values are less than the significance level, the shift in the mean is low and/or CLES is close to 0.5.

4.3 Effect of Scheduling Configurations

Rows 10 through 15 in Table A.2 show Brunner-Munzel [21] tests for comparing *FIFO* with *Default* scheduling configuration for Experiment 1. Rows 16 through 21 in Table A.2 show Brunner-Munzel tests for comparing *RR* with *Default* scheduling configuration for Experiment 1. The p-values (< 0.05) indicate that *FIFO* and *RR* show improvement in jitter in periodicity compared to *Default* scheduling configuration. The shift in mean due to *FIFO* and *RR* is in the range of milliseconds for *Ruby* and *Ruby:JIT* language configurations. The shift in mean due to *FIFO* and *RR* is in the range of tens or hundreds of microseconds for *C* and *Ruby:C* language configurations.

For Experiment 2 and Experiment 3, the improvement in temporal characteristics is not as evident as in the case of Experiment 1. Brunner-Munzel tests (rows 10 through 21 in

System Call	Metasploit-Ruby-$*$	Metasploit-Ruby:C-$*$
socket	1	1
ioctl	1	1
bind	1	1
write	1	1
close	1	1
fcntl	1	0
fstat	1	0
getsockname	1	0
setsockopt	1	0
futex	25	0
connect	1	0
getpid	11	0
getsockopt	2	0
ppoll	3	0
select	2	0
sendto	1	0
recvfrom	1	0
read	2	0
getpeername	1	0
shutdown	1	0

Table 4.1: List of system calls made along with the number of times the system call was made to transmit a CAN message for *Metasploit-Ruby-$*$* and *Metasploit-Ruby:C-$*$*

Table A.4 and Table A.6) show evidence in favour of the alternative hypothesis, i.e., *FIFO* and *RR* improve temporal characteristics, in some cases and in favour of null hypothesis in other cases. The shift in the mean transmission and round-trip latency due to the use of *FIFO* or *RR* instead of *Default* scheduling configuration are small compared to the shift in mean due to the use of *Standalone* framework configuration or use of *Ruby:C* language configuration. The authors monitored the number of involuntary context switches (available in the \proc\PID\status file in Linux) and the amount of time the processes spent in the ready queue waiting for CPU time (available in the \proc\PID\schedstat file in Linux). The data in Table 4.2 shows that there was indeed a reduction in involuntary context switches and the time waited by the process in the ready queue. For a single iteration, the reduction in the wait time is nearly 4 microseconds. Considering that the CAN message transmission latency for *Metasploit-Ruby-$*$* is in the range of tens of mil-

Scheduling Configuration	No. of Involuntary Context Switches	Wait Time in Ready Queue (microseconds)
Default	84	7184
FIFO	1	2976
RR	4	3091

Table 4.2: Number of involuntary context switches and wait time of a process in the ready queue for *Metasploit-Ruby-∗* program configurations for the program shown in Pseudocode 2 for 1000 iterations

liseconds, the improvement due to scheduling configuration is not as evident as is in the case of improvement due to change in the use of framework configuration and language configuration.

4.4 Effect of Period on Periodicity Jitter

From Figure 4.1, mean jitter for periodic tasks implemented using *Metasploit-Ruby-∗* configurations is an order of magnitude higher compared to mean jitter in *Standalone-Ruby-∗* configurations. In *Metasploit-Ruby-∗* configurations, mean jitter is in the range of hundreds of milliseconds except for a period of 5.0s, for which the jitter is in the range of tens of milliseconds. For *Metasploit-Ruby:C-∗* configurations, the jitter is not so different compared to *Standalone-C-∗* or *Standalone-Ruby:C-∗* configurations. As discussed in Section 4.1, it was found that in *Metasploit-Ruby-∗* configurations, sleep is implemented in multiples of 200ms sleep durations. A sleep period of 0.1 seconds would result in an actual sleep period of at least 0.2 seconds. A sleep period of 0.21 seconds would instead result in a sleep period of at least 0.4 seconds. Another interesting behaviour is that the number of 0.2 seconds sleep blocks is sometimes more than the required number of 200ms sleep durations. For 0.1s and 0.5s sleep durations, the actual sleep durations were 0.2s and 0.6s. In the case of sleep durations of 1.0s, 2.0s, 10.0s and 20.0s, even though the sleep durations are multiples of 200ms sleep duration, there was an extra 200ms sleep call for the specified sleep durations. For the sleep duration of 5.0s, the number of 200ms sleep calls was as required, i.e. 25. Hence, the jitter in periodicity for 5.0s sleep duration is lower than for other sleep durations. This is also the reason the study uses jitter data for 5.0s sleep duration for analysis in the above sections.

Figure 4.5 shows a plot of mean jitter in periodic CAN message transmission *Standalone-Ruby-∗*, *Standalone-C-∗* and *Arduino* program configurations. *Standalone-Ruby-Default*

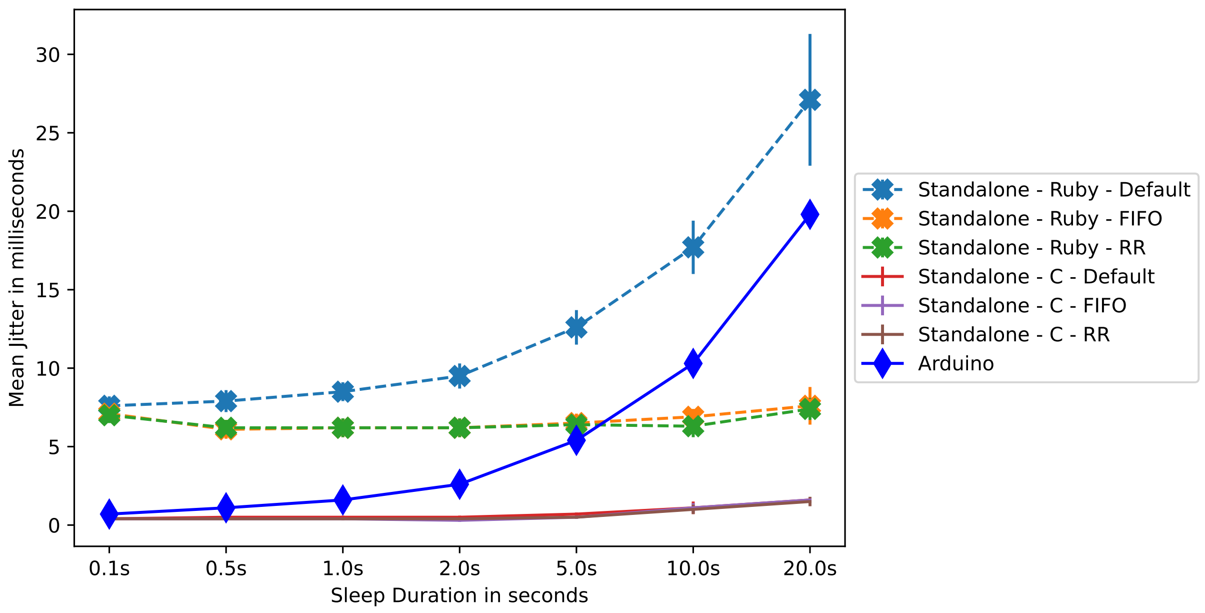

Figure 4.5: Graph showing mean jitter in periodic CAN message transmission for selected program configurations from Figure 4.1.

and *Arduino* program configurations show increasing mean jitter with the sleep duration (or the period). Figure 4.5 does not show such increasing mean jitter for *Standalone-Ruby-FIFO*, *Standalone-Ruby-RR* and *Standalone-C-∗* configurations. In the case of Ruby programming language, the *sleep* function call uses *ppoll* system call. On checking the time spent in the *ppoll* system call for various sleep durations using *strace* tool, *ppoll* system call shows similar increasing jitter with sleep duration. The *usleep* function in C programming language uses *clock_nanosleep* system call [6].

4.5 Temporal Characteristics of Arduino Uno

This section compares the temporal characteristics of *Arduino* with *Metasploit-Ruby:C-Default*. The section chooses *Metasploit-Ruby:C-Default* program configuration for comparison with *Arduino*. The Brunner-Munzel test [21] presented in Row 31 in Table A.2 and Table A.4 shows that *Arduino* has worse jitter in periodicity and worse transmission latency compared to *Metasploit-Ruby:C-Default* (p-value < 0.05). In the case of round-trip latency, the Brunner-Munzel test in Row 31 of Table A.6 shows that *Arduino* is better than

Metasploit-Ruby:C-Default.

Regarding the dependency of jitter in periodicity, from Figure 4.1, the study finds that the jitter in periodicity increases with the period and is worse compared to the *Metasploit-Ruby:C-Default* program configuration. To check the increasing jitter in Arduino Uno, the study checked the mean jitter for a period of 160 seconds for a sample size of 100. The study found the mean jitter for the period of 160 seconds to be 157.3 milliseconds. The study found the mean jitter values for the last three periods (5s, 10s and 20s) to be 5.4ms, 10.3ms, and 19.9ms, respectively. This data on the mean jitter for the period of 160 seconds is consistent with the data obtained for other values of the period and emphasizes the nature of increasing jitter with the value of the period.

Chapter 5

Case Study

This chapter explores the program configurations discussed in effect. The case studies done below demonstrate the effect of the configurations on the timing characteristics studied in the experiments in Section 3.4 and consequently on the success of the attacks. We analyze how the *Metasploit-Ruby-Default* configuration compares to the *Metasploit-Ruby:C-Default* configuration in actual testing/attack scenarios.

5.1 CAN-Flood Attack

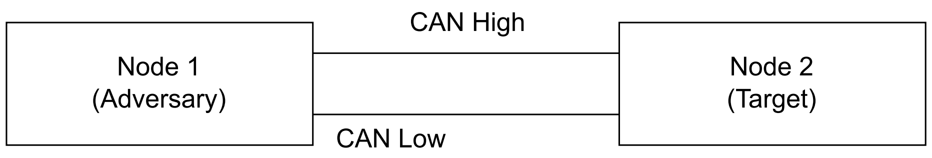

Figure 5.1: Setup for CAN-Flood attack with two CAN nodes

The thesis provided background information on the CAN-Flood attack in Section 2.4.1. To effectively cause Denial-of-Service to non-malicious nodes, the malicious node needs to increase the bus-load by sending messages with a low CAN ID. To monitor the bus-load due to a malicious node, the study designates a node as the adversary in a CAN Network, with only the adversary sending CAN messages. The study measures the bus-load in the CAN network. The setup is shown in Figure 5.1. On using the *Metasploit-Ruby-Default* configuration to execute the CAN-Flood attack [22, 10], the bus load increases from 0%

to 3%. On using the *Metasploit-Ruby:C-Default* configuration, the bus load increases from 0% to 79%. The study highlights the improvement offered by using C to implement the CAN-Flood attack compared to Ruby.

5.2 Bus-Off Attack

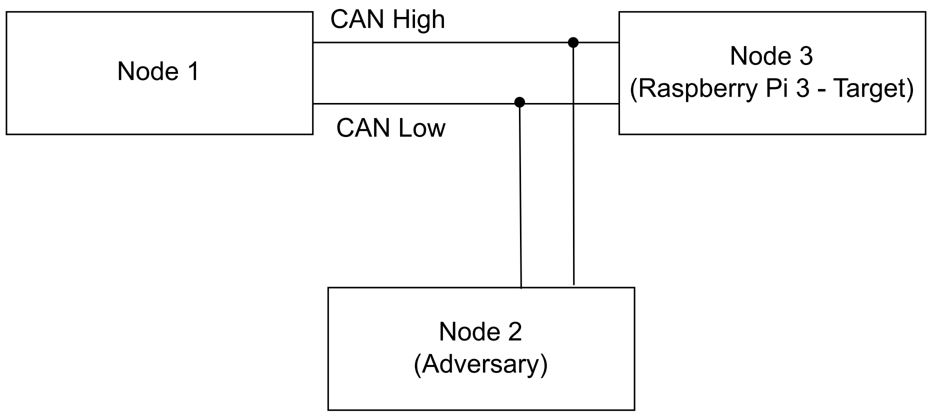

Figure 5.2: Setup for Bus-Off attack with three CAN nodes

The thesis provided background on Bus-Off attack [24] in Section 2.4.2. The setup shown in Figure 5.2 is used to evaluate the Bus-Off attack. Node 1 uses *Metasploit-Ruby:C-Default* to transmit the preceded messages periodically. Node 2 uses Metasploit to transmit the attack messages, with the first preceded message sent by Node 1 acting as the trigger message. Node 3 (Raspberry Pi 3) is the target of the attack. Node 3 transmits the target message on the reception of the first preceded message from Node 1.

For Node 3, the round-trip latency is estimated to be 541μs with a jitter of 24μs. For Node 2, the study chooses the *Metasploit-Ruby:C-Default* configuration. The round-trip latency for Node 2 is then 2134μs with a jitter of 517μs (from the data shown in Table A.5). Since there is a difference in the round-trip latency of the target node and the adversary node, Node 1 must keep the bus busy such that the target message and the attack message synchronize. Using two standard deviations as the upper bound on the round-trip latency, $D_{target,max}$ is 589μs and $D_{attack,max}$ is 3168 μs. Considering $t_{trigger}$ to be 220μs and $t_{precede}$ to be 248μs, the number of preceded messages needed is 12 (using Equation 2.1). In the Bus-Off attack experiment discussed, in 100 attempts, on average, 611 attack messages were needed before the target node entered the *Bus-Off* state. Using

Metasploit-Ruby-Default configuration as the adversary, the difference in round-trip latency between adversary (mean latency = $18361\mu s$ and standard deviation = $1443\mu s$, from Table A.5) and the target node is so high that the bus-off attack does not succeed. When the Bus-Off attack experiment was performed with Node 2 using Raspberry Pi 3, in 100 attempts, on average, 56 attack messages were needed to force the target node into the bus-off state. This Bus-Off attack experiment highlights the drawback of the high latency in a Metasploit-based adversary compared to the target node.

Chapter 6

Microcontroller Integration with Metasploit

A computer-based test platform has the overhead of an operating system kernel and has to compete with many processes running in the system for CPU, memory and I/O. A microcontroller-based test platform can provide better temporal characteristics for tests compared to a computer-based test platform (refer to Section 4.5). This chapter discusses how the Metasploit Framework can be used in coordination with a microcontroller-based test platform.

6.1 Proposed Framework

The Metasploit Framework provides a repository or database of vulnerabilities, exploits, post-exploitation modules, etc. A test or attack that can or needs to be implemented using a microcontroller can also be stored as a module in the Metasploit Framework. Once all the modules which are run in a microcontroller are available for use with the Metasploit Framework, a test module is written in order to load the microcontroller program onto the microcontroller and execute the test.

A test module has the following interfaces - *info_module*, *load_test*, *start_test*, *info_test* and *stop_test*. Figure 6.1 shows the interfaces in the test module, which is an integration of a Metasploit Module and a Microcontroller Test Platform. The *info_module* interface provides information about the module and about what the test does. The *run_module* interface starts the module. The Metasploit Module (as part of the Metasploit Framework)

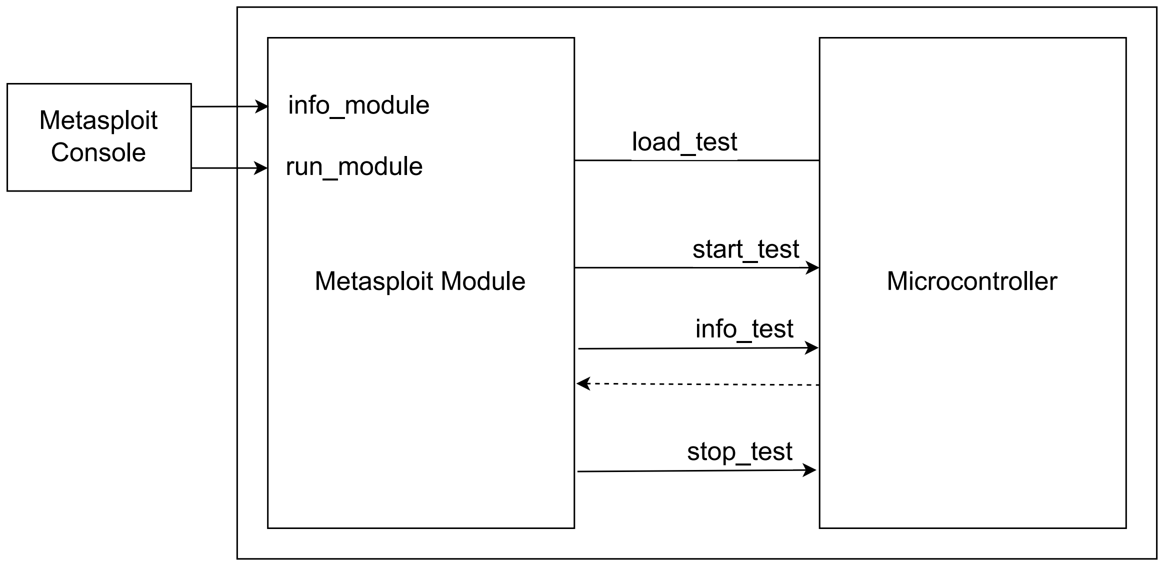

Figure 6.1: Proposed framework for integration of microcontroller test platform with Metasploit module

implements the interfaces *info_module* and *run_module*. The *load_test* interface allows the Metasploit Module to load the test onto the microcontroller. The interface can be realized using a bootloader where the bootloader waits for the Metasploit Module to send the actual test so that the bootloader can initiate the test. The *load_test* interface can also be realized using a flashing tool which flashes the program binary onto the microcontroller. The microcontroller implements the *start_test*, *info_test* and *stop_test* interfaces. The Metasploit Module utilizes the interfaces to control the test execution using *start_test* and *stop_test*. The Metasploit Module utilizes the interface *info_test* to gain information about the status of the execution of the test.

6.2 A Realization of the Proposed Framework

In this study, an Arduino Uno has been utilized as the microcontroller for integration with Metasploit Module. A Metasploit Module implements the *info_module* and *run_module* interfaces. The *run_module* interface builds the test, loads the test onto the microcontroller and invokes *start_test* interface to start the test. The *arduino_cli* tool is used to build or compile the Arduino program (or sketch) and flash the binary onto the Arduino board.

The Metasploit Module and the Arduino Program communicate over Universal Asynchronous Receiver-Transmitter (UART) port. The Arduino Program registers an interrupt on the UART port. The program waits for commands from the Metasploit Module from UART port to start, stop or return information of the test. The Arduino Program acts accordingly on receiving commands from the Metasploit Module. When the Metasploit Module invokes the *info_test* interface, the module waits for a response ending in a newline character from the Arduino Program. Pseudocode 4 shows the pseudocode for the realization of the proposed framework using Arduino Uno. The program handles the commands related to the interfaces *start_test*, *stop_test* and *info_test* in the *command_handler* function.

Pseudocode 4: Pseudocode showing the realization of the proposed framework using Arduino Uno

```
function setup():
    register_UART_interrupt()
    other_initializations()
function uartIsr():
    var_command ← uart_input
function mainloop():
    if (new command received):
        command_handler(var_command)
    if (start condition met):
        run_test()
```

Chapter 7

Conclusion

It is necessary to ensure the security of both cyber and physical components in a CPS. The Metasploit Framework extended the support for testing cyber-physical systems by introducing the Hardware Bridge API. The authors argue that the framework needs to be evaluated considering the real-time constraints of the tests designed for CPS. This study determined the jitter in the periodic transmission of CAN messages, the latency transmission of a CAN message and the round-trip latency of a CAN message for different combinations of factors - use of framework, language and scheduling configuration. The study showed that the methods and APIs provided by Metasploit perform significantly more operations under the hood compared to a C implementation of the same functionality. Hence, using Metasploit-provided APIs can add significant latency and jitter to the task under consideration (in this case, CAN message transmission and reception).

For the periodic transmission of CAN messages, the study found some interesting observations. First, the results showed increasing jitter in the period with respect to actual sleep duration in the *ppoll* system call used in the sleep function in Ruby when executed with the SCHED_OTHER scheduling policy. The study found similar increasing behaviour in jitter in the case of the delay function in Arduino Uno as well. Second, the study showed that the sleep duration took effect in multiples of 200ms in Metasploit.

The results show that the improvement due to the real-time scheduling policies was small compared to the improvement due to the use of C programming language for implementing the tests in case of transmission and round-trip latency. The number of involuntary context switches and wait time in the ready queue were reduced due to the use of real-time scheduling policies with high process priorities. Enabling JIT compilation in Ruby during the execution of a Ruby script did not bring any discernible improvement in the temporal characteristics for the task of interacting with the CAN network. The thesis compared the temporal characteristics of a microcontroller using Arduino Uno as an example with a Metasploit module. The thesis proposed a framework in order to integrate tests that are executed on a microcontroller.

Each cyber-physical system is different, and each test can have different timing constraints. For tests with real-time constraints, the temporal characteristics need to be analyzed for each system-under-test, test and platform used for the test. Regardless, this study has shown the extent of the effect of different factors on the temporal characteristics of tests implemented as a Metasploit module. This study has shown that the latency and jitter in the transmission and reception of CAN messages using the Metasploit Framework are improved significantly by implementing the functionality in C as a shared library. This study showed the extent of the impact of using real-time scheduling policies and JIT compilation on the temporal characteristics of the tests.

References

[1] Arduino Programming Language Reference. https://www.arduino.cc/reference/en/.

[2] Documentation for delay function in Arduino. https://www.arduino.cc/reference/en/language/functions/time/delay/.

[3] Documentation for sleep function in Ruby. https://apidock.com/ruby/Kernel/sleep.

[4] Documentation for SocketCAN library. https://docs.kernel.org/networking/can.html.

[5] Documentation for usleep function in C. https://man7.org/linux/man-pages/man3/usleep.3.html.

[6] Documentation on clock_nanosleep system call. https://linux.die.net/man/2/clock_nanosleep.

[7] Documentation on ppoll system call. https://linux.die.net/man/2/ppoll.

[8] Metasploit Framework: Automotive Post-Exploitation Modules. https://github.com/rapid7/metasploit-framework/tree/master/documentation/modules/post/hardware/automotive.

[9] Metasploit Framework: sleep function. https://github.com/rapid7/metasploit-framework/blob/b4991a97d02572a323470aa0906e934cce1b7843/lib/rex.rb#L102.

[10] Metasploit Module: CAN-Flood. https://github.com/rapid7/metasploit-framework/blob/master/modules/post/hardware/automotive/can_flood.rb.

[11] SocketCAN Userspace Utilities and Tools. https://github.com/linux-can/can-utils.

[12] Source code for APIs provided by Metasploit for Interaction with CAN interface. https://github.com/rapid7/metasploit-framework/blob/master/modules/auxiliary/server/local_hwbridge.rb.

[13] Metasploit - Penetration Testing Software, Website: https://www.metasploit.com/. https://www.metasploit.com/, 2022.

[14] Speed comparison of programming languages. https://niklas-heer.github.io/speed-comparison/, 2023.

[15] Huthaifa Al-Omari, Francis Wolff, Christos Papachristou, and David McIntyre. An Improved Algorithm to Smooth Delay Jitter in Cyber-Physical Systems. In *2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing*, pages 81–86, 2009.

[16] Arduino. Arduino Uno R3 - Product Reference Manual. https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf, 2023.

[17] John Aycock. A Brief History of Just-in-Time. *ACM Comput. Surv.*, 35(2):97–113, June 2003.

[18] G. Bollella and K. Jeffay. Support for real-time computing within general purpose operating systems-supporting co-resident operating systems. In *Proceedings Real-Time Technology and Applications Symposium*, pages 4–14, 1995.

[19] Robert Bosch et al. Can specification version 2.0. *Robert Bosch GmbH, Postfach*, 50, 1991.

[20] 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, 2018.

[21] Edgar Brunner and Ullrich Munzel. The nonparametric Behrens-Fisher problem: asymptotic theory and a small-sample approximation. *Biometrical Journal: Journal of Mathematical Methods in Biosciences*, 42(1):17–25, 2000.

[22] Paul Carsten, Todd R. Andel, Mark Yampolskiy, and Jeffrey T. McDonald. In-Vehicle Networks: Attacks, Vulnerabilities, and Proposed Solutions. In *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, CISR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[23] A. Cervin. Improved Scheduling of Control Tasks. In *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*, pages 4–10, 1999.

[24] 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, 2016.

[25] Jacob Cohen. *Statistical power analysis for the behavioral sciences.* Academic press, 2013.

[26] 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.

[27] Abhishek Dubey, Gabor Karsai, and Sherif Abdelwahed. Compensating for Timing Jitter in Computing Systems with General-Purpose Operating Systems. In *2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 55–62, 2009.

[28] Olive Jean Dunn. Multiple comparisons among means. *Journal of the American statistical association*, 56(293):52–64, 1961.

[29] Miro Enev, Alex Takakuwa, Karl Koscher, and Tadayoshi Kohno. Automobile Driver Fingerprinting. *Proceedings on Privacy Enhancing Technologies*, 2016:34 – 50, 2016.

[30] Konrad Etschberger, Roman Hofmann, Joachim Stolberg, Christian Schlegel, and Stefan Weiher. *Controller area network: basics, protocols, chips and applications.* IXXAT Automation, 2001.

[31] Thomas Fischl. USBtin - USB to CAN interface. https://www.fischl.de/usbtin/, 2016.

[32] Rory A. Fisher. Studies in crop variation. I. An examination of the yield of dressed grain from Broadbalk. *The Journal of Agricultural Science*, 11:107 – 135, 1921.

[33] Rory A. Fisher and Winifred A. Mackenzie. Studies in crop variation. ii. the manurial response of different potato varieties. *The Journal of Agricultural Science*, 13:311 – 320, 1923.

[34] Michael A Fligner and George E Policello. Robust rank procedures for the Behrens-Fisher problem. *Journal of the American Statistical Association*, 76(373):162–168, 1981.

[35] Dip Goswami, Reinhard Schneider, Alejandro Masrur, Martin Lukasiewycz, Samarjit Chakraborty, Harald Voit, and Anuradha Annaswamy. Challenges in Automotive Cyber-Physical Systems Design. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 346–354, 2012.

[36] Bogdan Groza and Pal-Stefan Murvay. Security Solutions for the Controller Area Network: Bringing Authentication to In-Vehicle Networks. *IEEE Vehicular Technology Magazine*, 13(1):40–47, 2018.

[37] Roland E. Haas and Dietmar P. F. Möller. Automotive Connectivity, Cyber Attack Scenarios and Automotive Cyber Security. In *2017 IEEE International Conference on Electro Information Technology (EIT)*, pages 635–639, 2017.

[38] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[39] Steve Corrigan HPL. Introduction to the controller area network (CAN). *Application Report SLOA101*, pages 1–17, 2002.

[40] Tingting Hu. Deterministic and flexible communication for real-time embedded systems. 2015.

[41] Ryusuke Imai and Ryogo Kubo. Introducing Jitter Buffers in Networked Control Systems With Communication Disturbance Observer Under Time-Varying Communication Delays. In *IECON 2015 - 41st Annual Conference of the IEEE Industrial Electronics Society*, pages 002956–002961, 2015.

[42] Microchip Technology Inc. MCP2551 - High-Speed CAN Transceiver. https://ww1.microchip.com/downloads/en/devicedoc/20001667g.pdf, 2016.

[43] Microchip Technology Inc. MCP2515 - Stand-Alone CAN Controller with SPI Interface. https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf, 2019.

[44] Julian D Karch. Psychologists should use Brunner-Munzel's instead of Mann-Whitney's U test as the default nonparametric procedure. *Advances in Methods and Practices in Psychological Science*, 4(2):2515245921999602, 2021.

[45] Stamatis Karnouskos. Stuxnet Worm Impact on Industrial Cyber-Physical System Security. In *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*, pages 4490–4494, 2011.

[46] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*, chapter 35, pages 733–752. No Starch Press, USA, 1 edition, 2010.

[47] 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 *2010 IEEE symposium on security and privacy*, pages 447–462. IEEE, 2010.

[48] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, 47(260):583–621, 1952.

[49] Howard Levene. Robust tests for the equality of variance in: Olkin, i., contributions to probability and statistics: Essays in honor of harold hotelling, 1960.

[50] Jiajia Liu, Shubin Zhang, Wen Sun, and Yongpeng Shi. In-Vehicle Network Attacks and Countermeasures: Challenges and Future Directions. *IEEE Network*, 31(5):50–58, 2017.

[51] M. Lluesma, A. Cervin, P. Balbastre, I. Ripoll, and A. Crespo. Jitter Evaluation of Real-Time Control Systems. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*, pages 257–260, 2006.

[52] Kenneth O McGraw and Seok P Wong. A common language effect size statistic. *Psychological bulletin*, 111(2):361, 1992.

[53] Wayne Meissner. ffi — RubyGems.org. https://rubygems.org/gems/ffi/, 2021.

[54] Charlie Miller and Chris Valasek. Remote Exploitation of an Unaltered Passenger Vehicle. *Black Hat USA*, 2015(S 91), 2015.

[55] Sandeep Mistry. Source code for CAN Library for Arduino. https://github.com/sandeepmistry/arduino-CAN.

[56] Subhojeet Mukherjee, Hossein Shirazi, Indrakshi Ray, Jeremy S. Daily, and Rose F. Gamble. Practical DoS Attacks on Embedded Networks in Commercial Vehicles. In *ICISS*, 2016.

[57] Sebastian Nanz and Carlo A Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788. IEEE, 2015.

[58] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. A Stealth, Selective, Link-Layer Denial-of-Service Attack Against Automotive Networks. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 185–206, Cham, 2017. Springer International Publishing.

[59] Andrea Palanca, Eric Evenchick, Federico Maggi, and Stefano Zanero. A Stealth, Selective, Link-Layer Denial-of-Service Attack Against Automotive Networks. In *DIMVA*, 2017.

[60] PEAK-System. PCAN-USB: CAN Interface for USB. https://www.peak-system.com/PCAN-USB.199.0.html?&L=1, 2022.

[61] Enrico Pozzobon, de, and Nils Weiss. A Survey on Media Access Solutions for CAN Penetration Testing. 2018.

[62] K. Ramamritham, Chia Shen, O. Gonzalez, S. Sen, and S. Shirgurkar. Using windows nt for real-time applications: experimental observations and recommendations. In *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245)*, pages 102–111, 1998.

[63] Alexandre S. Roque, Nasser Jazdi, Edisons P. Freitas, and Carlos E. Pereira. Performance Analysis of In-Vehicle Distributed Control Systems Applying a Real-Time Jitter Monitor. In *2020 IEEE 18th International Conference on Industrial Informatics (INDIN)*, volume 1, pages 663–668, 2020.

[64] Nipun Jaswal Sagar Rahalkar. *Metasploit Revealed: Secrets of the Expert Pentester.* Packt Publishing Ltd, 2017.

[65] D. Seto, J.P. Lehoczky, and Liu Sha. Task Period Selection and Schedulability in Real-Time Systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 188–198, 1998.

[66] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[67] Sulav Lal Shrestha, Taylor Lee, and Sebastian Fischmeister. Metasploit for cyber-physical security testing with real-time constraints. In *International Conference on Science of Cyber Security*, pages 260–275. Springer, 2022.

[68] Craig Smith. Exiting the Matrix: Introducing Metasploit's Hardware Bridge. `https://www.rapid7.com/blog/post/2017/02/02/exiting-the-matrix/`, 2017.

[69] Michal Sojka, Pavel Píša, Martin Petera, Ondřej Špinka, and Zdeněk Hanzálek. A comparison of Linux CAN drivers and their applications. In *International Symposium on Industrial Embedded System (SIES)*, pages 18–27, 2010.

[70] Student. The probable error of a mean. *Biometrika*, 6(1):1–25, 1908.

[71] Bernard L Welch. The generalization of 'STUDENT'S'problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[72] Bernard Lewis Welch. On the comparison of several mean values: an alternative approach. *Biometrika*, 38(3/4):330–336, 1951.

[73] F Wilcoxon. Individual comparisons by ranking methods. Biom Bull 1 (6): 80–83, 1945.

APPENDICES

Appendix A

Data Summary and Statistical Tests

The tables below show the data summary for the experiments in Section 3.4 and the results of Brunner-Munzel tests between various program configurations. Tables A.1, A.3 and A.5 show the mean, standard deviation, median and Inter-Quartile Range (IQR) for (i) the jitter in periodic CAN message transmission with period of 5.0s, (ii) the transmission latency, and (iii) the round-trip latency for different program configurations respectively. The sample size for each program configuration is 1000.

Table A.2, Table A.4, and Table A.6 show the p-values for Brunner-Munzel Test [21] between the program configuration in PrgConfig 1 column and the program configuration in PrgConfig 2 column. Each test between PrgConfig 1 and PrgConfig 2 has the alternative hypothesis that when a random sample is chosen from PrgConfig 1 and one from PrgConfig 2, the random sample from PrgConfig 1 has a smaller value compared to the sample from PrgConfig 2. The tables also show CLES [52], which is the probability that the random sample from PrgConfig 1 has a smaller value than the random sample from PrgConfig 2. The last column "$\Delta\mu$" shows the difference in mean between PrgConfig 1 and PrgConfig 2 as the measure of unstandardized effect size. Table A.4 shows the tests between the program configurations for data in Table A.3. Table A.6 shows the p-value for the Brunner-Munzel test, CLES and the difference in means for the two program configurations for data in Table A.5. The p-values shown in the tables are p-values corrected using Holm's method [38] for multiple tests.

In the tables, *MSF* stands for *Metasploit* and *STN* stands for *Standalone*. The framework configurations have been shortened to fit the tables within the page boundaries.

Program Configuration	**mean** (μs)	**sd** (μs)	**median** (μs)	**IQR**(μs)
Metasploit - Ruby - Default	33755.8	2366.3	34330.0	1450.0
Metasploit - Ruby - FIFO	28552.7	2611.5	29340.0	1748.8
Metasploit - Ruby - RR	28523.3	2400.5	29110.0	1550.0
Metasploit - Ruby:C - Default	699.4	161.5	750.0	155.0
Metasploit - Ruby:C - FIFO	612.4	165.4	660.0	220.0
Metasploit - Ruby:C - RR	670.6	120.6	700.0	120.0
Standalone - Ruby - Default	12600.3	1080.1	12910.0	429.2
Standalone - Ruby - FIFO	6478.7	554.3	6660.0	150.0
Standalone - Ruby - RR	6442.4	522.3	6650.0	343.5
Standalone - Ruby:JIT - Default	12577.0	1170.9	13000.0	230.5
Standalone - Ruby:JIT- FIFO	6465.7	558.4	6660.0	170.0
Standalone - Ruby:JIT- RR	6476.7	504.4	6656.0	190.0
Standalone - Ruby:C - Default	713.7	152.1	740.0	144.0
Standalone - Ruby:C - FIFO	672.5	129.9	710.0	138.5
Standalone - Ruby:C - RR	683.6	155.0	710.0	170.0
Standalone - C - Default	742.8	133.4	764.0	140.0
Standalone - C - FIFO	482.4	107.0	440.0	100.0
Standalone - C - RR	482.1	108.7	440.0	126.2
Arduino	5388.4	8.0	5390.0	10.0

Table A.1: Data summary for Experiment 1 (period 5.0s)

	PrgConfig 1	**PrgConfig 2**	**p-value**	**CLES**	$\Delta\mu$ (μs)
1	MSF-Ruby:C-Default	MSF-Ruby-Default	0.0	1.0	-33056.4
2	MSF-Ruby:C-Default	MSF-Ruby-FIFO	0.0	1.0	-27853.3
3	MSF-Ruby:C-Default	MSF-Ruby-RR	0.0	1.0	-27823.9
4	MSF-Ruby:C-Default	STN-Ruby-Default	0.0	1.0	-11900.9
5	MSF-Ruby:C-Default	STN-Ruby-FIFO	0.0	1.0	-5779.3
6	MSF-Ruby:C-Default	STN-Ruby-RR	0.0	1.0	-5743.0
7	MSF-Ruby:C-Default	STN-Ruby:JIT-Default	0.0	1.0	-11877.6
8	MSF-Ruby:C-Default	STN-Ruby:JIT-FIFO	0.0	1.0	-5766.3
9	MSF-Ruby:C-Default	STN-Ruby:JIT-RR	0.0	1.0	-5777.3
10	MSF-Ruby-FIFO	MSF-Ruby-Default	0.0	0.93	-5203.1
11	MSF-Ruby:C-FIFO	MSF-Ruby:C-Default	0.0	0.69	-87.0
12	STN-Ruby-FIFO	STN-Ruby-Default	0.0	1.0	-6121.6
13	STN-Ruby:JIT-FIFO	STN-Ruby:JIT-Default	0.0	1.0	-6111.3
14	STN-Ruby:C-FIFO	STN-Ruby:C-Default	0.0	0.6	-41.2
15	STN-C-FIFO	STN-C-Default	0.0	0.92	-260.4
16	MSF-Ruby-RR	MSF-Ruby-Default	0.0	0.93	-5232.5
17	MSF-Ruby:C-RR	MSF-Ruby:C-Default	0.0	0.63	-28.8
18	STN-Ruby-RR	STN-Ruby-Default	0.0	1.0	-6157.9
19	STN-Ruby:JIT-RR	STN-Ruby:JIT-Default	0.0	1.0	-6100.3
20	STN-Ruby:C-RR	STN-Ruby:C-Default	0.0	0.56	-30.1
21	STN-C-RR	STN-C-Default	0.0	0.92	-260.7
22	STN-Ruby-Default	MSF-Ruby-Default	0.0	1.0	-21155.5
23	STN-Ruby-FIFO	MSF-Ruby-FIFO	0.0	1.0	-22074.0
24	STN-Ruby-RR	MSF-Ruby-RR	0.0	1.0	-22080.9
25	STN-C-Default	MSF-Ruby:C-Default	1.0	0.44	43.4
26	STN-C-FIFO	MSF-Ruby:C-FIFO	0.0	0.75	-130.0
27	STN-C-RR	MSF-Ruby:C-RR	0.0	0.86	-188.5
28	STN-Ruby:JIT-Default	STN-Ruby-Default	1.0	0.46	-23.3
29	STN-Ruby:JIT-FIFO	STN-Ruby-FIFO	1.0	0.5	-13.0
30	STN-Ruby:JIT-RR	STN-Ruby-RR	1.0	0.47	34.3
31	Arduino	MSF-Ruby:C-Default	1.0	0.0	4689.0

Table A.2: Table showing p-value for Brunner-Munzel test, CLES and difference in mean (μs) for two sample distributions for Experiment 1 (period 5.0s)

Program Configuration	**mean** (μs)	**sd** (μs)	**median** (μs)	**IQR** (μs)
Metasploit - Ruby - Default	16911.9	2937.2	17518.0	845.9
Metasploit - Ruby - FIFO	16641.4	2713.5	17076.0	799.9
Metasploit - Ruby - RR	16851.7	2445.3	17178.3	802.7
Metasploit - Ruby:C - Default	10.1	0.2	10.0	0.1
Metasploit - Ruby:C - FIFO	10.1	0.2	10.0	0.1
Metasploit - Ruby:C - RR	10.1	0.2	10.0	0.1
Standalone - Ruby - Default	5364.5	682.5	5500.4	57.9
Standalone - Ruby - FIFO	5106.4	566.6	5190.4	44.6
Standalone - Ruby - RR	5022.6	667.8	5148.8	46.2
Standalone - Ruby:JIT - Default	5387.4	626.1	5506.3	54.4
Standalone - Ruby:JIT - FIFO	5078.1	636.5	5191.5	44.0
Standalone - Ruby:JIT - RR	5127.0	518.7	5200.0	42.0
Standalone - Ruby:C - Default	10.1	0.2	10.0	0.1
Standalone - Ruby:C - FIFO	10.1	0.2	10.0	0.1
Standalone - Ruby:C - RR	10.1	0.3	10.0	0.1
Standalone - C - Default	10.1	0.2	10.0	0.1
Standalone - C - FIFO	10.1	0.2	10.0	0.1
Standalone - C - RR	10.1	0.3	10.0	0.1
Arduino	366.1	3.3	368.1	6.0

Table A.3: Data summary for Experiment 2

	PrgConfig 1	**PrgConfig 2**	**p-value**	**CLES**	$\Delta\mu$ (μs)
1	MSF-Ruby:C-Default	MSF-Ruby-Default	0.0	1.0	-16901.8
2	MSF-Ruby:C-Default	MSF-Ruby-FIFO	0.0	1.0	-16631.3
3	MSF-Ruby:C-Default	MSF-Ruby-RR	0.0	1.0	-16841.6
4	MSF-Ruby:C-Default	STN-Ruby-Default	0.0	1.0	-5354.4
5	MSF-Ruby:C-Default	STN-Ruby-FIFO	0.0	1.0	-5096.3
6	MSF-Ruby:C-Default	STN-Ruby-RR	0.0	1.0	-5012.5
7	MSF-Ruby:C-Default	STN-Ruby:JIT-Default	0.0	1.0	-5377.3
8	MSF-Ruby:C-Default	STN-Ruby:JIT-FIFO	0.0	1.0	-5068.0
9	MSF-Ruby:C-Default	STN-Ruby:JIT-RR	0.0	1.0	-5116.9
10	MSF-Ruby-FIFO	MSF-Ruby-Default	0.0	0.65	-270.5
11	MSF-Ruby:C-FIFO	MSF-Ruby:C-Default	1.0	0.51	0.0
12	STN-Ruby-FIFO	STN-Ruby-Default	0.0	0.95	-258.1
13	STN-Ruby:JIT-FIFO	STN-Ruby:JIT-Default	0.0	0.96	-309.3
14	STN-Ruby:C-FIFO	STN-Ruby:C-Default	0.01	0.54	0.0
15	STN-C-FIFO	STN-C-Default	1.0	0.49	0.0
16	MSF-Ruby-RR	MSF-Ruby-Default	0.0	0.61	-60.2
17	MSF-Ruby:C-RR	MSF-Ruby:C-Default	1.0	0.51	0.0
18	STN-Ruby-RR	STN-Ruby-Default	0.0	0.95	-341.9
19	STN-Ruby:JIT-RR	STN-Ruby:JIT-Default	0.0	0.96	-260.4
20	STN-Ruby:C-RR	STN-Ruby:C-Default	1.0	0.49	0.0
21	STN-C-RR	STN-C-Default	1.0	0.5	0.0
22	STN-Ruby-Default	MSF-Ruby-Default	0.0	1.0	-11547.4
23	STN-Ruby-FIFO	MSF-Ruby-FIFO	0.0	1.0	-11535.0
24	STN-Ruby-RR	MSF-Ruby-RR	0.0	1.0	-11829.1
25	STN-C-Default	MSF-Ruby:C-Default	1.0	0.51	0.0
26	STN-C-FIFO	MSF-Ruby:C-FIFO	1.0	0.49	0.0
27	STN-C-RR	MSF-Ruby:C-RR	1.0	0.49	0.0
28	STN-Ruby:JIT-Default	STN-Ruby-Default	1.0	0.46	22.9
29	STN-Ruby:JIT-FIFO	STN-Ruby-FIFO	1.0	0.51	-28.3
30	STN-Ruby:JIT-RR	STN-Ruby-RR	1.0	0.15	104.4
31	Arduino	MSF-Ruby:C-Default	1.0	0.0	356.0

Table A.4: Table showing p-value for Brunner-Munzel test, CLES and difference in mean (μs) for two sample distributions for Experiment 2

Program Configuration	**mean** (μs)	**sd** (μs)	**median** (μs)	**IQR** (μs)
Metasploit - Ruby - Default	18361.4	1442.7	18637.5	1026.5
Metasploit - Ruby - FIFO	18084.6	5515.7	18635.5	1029.5
Metasploit - Ruby - RR	17990.6	5784.8	18797.5	1085.8
Metasploit - Ruby:C - Default	2134.3	516.7	1989.1	1003.0
Metasploit - Ruby:C - FIFO	2191.5	504.1	2197.1	986.8
Metasploit - Ruby:C - RR	2195.3	511.2	2277.5	1000.2
Standalone - Ruby - Default	8661.4	989.3	8680.3	1275.2
Standalone - Ruby - FIFO	8781.5	831.4	8830.8	1021.7
Standalone - Ruby - RR	8778.7	799.5	8850.8	984.9
Standalone - Ruby:JIT - Default	9246.3	695.5	9109.8	976.9
Standalone - Ruby:JIT - FIFO	8711.6	890.2	8703.8	1179.9
Standalone - Ruby:JIT - RR	8732.5	860.5	8712.7	1223.8
Standalone - Ruby:C - Default	2321.5	475.9	2658.2	915.8
Standalone - Ruby:C - FIFO	2141.9	516.7	2064.1	1012.9
Standalone - Ruby:C - RR	2148.4	511.9	1918.0	1007.8
Standalone - C - Default	2281.2	497.5	2656.1	940.5
Standalone - C - FIFO	2207.0	490.7	2396.0	958.5
Standalone - C - RR	2249.3	483.5	2473.2	931.5
Arduino	595.4	7.3	596.0	11.0

Table A.5: Data summary for Experiment 3

	PrgConfig 1	**PrgConfig 2**	**p-value**	**CLES**	$\Delta\mu$ (μs)
1	MSF-Ruby:C-Default	MSF-Ruby-Default	0.0	1.0	-16227.1
2	MSF-Ruby:C-Default	MSF-Ruby-FIFO	0.0	1.0	-15950.3
3	MSF-Ruby:C-Default	MSF-Ruby-RR	0.0	1.0	-15856.3
4	MSF-Ruby:C-Default	STN-Ruby-Default	0.0	1.0	-6527.1
5	MSF-Ruby:C-Default	STN-Ruby-FIFO	0.0	1.0	-6647.2
6	MSF-Ruby:C-Default	STN-Ruby-RR	0.0	1.0	-6644.4
7	MSF-Ruby:C-Default	STN-Ruby:JIT-Default	0.0	1.0	-7112.0
8	MSF-Ruby:C-Default	STN-Ruby:JIT-FIFO	0.0	1.0	-6577.3
9	MSF-Ruby:C-Default	STN-Ruby:JIT-RR	0.0	1.0	-6598.2
10	MSF-Ruby-FIFO	MSF-Ruby-Default	0.8	0.52	-276.8
11	MSF-Ruby:C-FIFO	MSF-Ruby:C-Default	1.0	0.44	57.2
12	STN-Ruby-FIFO	STN-Ruby-Default	1.0	0.48	120.1
13	STN-Ruby:JIT-FIFO	STN-Ruby:JIT-Default	0.0	0.75	-534.7
14	STN-Ruby:C-FIFO	STN-Ruby:C-Default	0.0	0.58	-179.6
15	STN-C-FIFO	STN-C-Default	0.0	0.6	-74.2
16	MSF-Ruby-RR	MSF-Ruby-Default	1.0	0.49	-370.8
17	MSF-Ruby:C-RR	MSF-Ruby:C-Default	1.0	0.44	61.0
18	STN-Ruby-RR	STN-Ruby-Default	1.0	0.48	117.3
19	STN-Ruby:JIT-RR	STN-Ruby:JIT-Default	0.0	0.74	-513.8
20	STN-Ruby:C-RR	STN-Ruby:C-Default	0.0	0.57	-173.1
21	STN-C-RR	STN-C-Default	1.0	0.49	-31.9
22	STN-Ruby-Default	MSF-Ruby-Default	0.0	1.0	-9700.0
23	STN-Ruby-FIFO	MSF-Ruby-FIFO	0.0	0.99	-9303.1
24	STN-Ruby-RR	MSF-Ruby-RR	0.0	1.0	-9211.9
25	STN-C-Default	MSF-Ruby:C-Default	1.0	0.41	146.9
26	STN-C-FIFO	MSF-Ruby:C-FIFO	0.0	0.57	15.5
27	STN-C-RR	MSF-Ruby:C-RR	1.0	0.46	54.0
28	STN-Ruby:JIT-Default	STN-Ruby-Default	1.0	0.25	584.9
29	STN-Ruby:JIT-FIFO	STN-Ruby-FIFO	1.0	0.5	-69.9
30	STN-Ruby:JIT-RR	STN-Ruby-RR	1.0	0.5	-46.2
31	Arduino	MSF-Ruby:C-Default	0.0	1.0	-1538.9

Table A.6: Table showing p-value for Brunner-Munzel test, CLES and difference in mean (μs) for two sample distributions for Experiment 3