Missouri S&T Library and Learning Resources

Scholars' Mine

Doctoral Dissertations | Student Theses and Dissertations

Summer 2020

Secure blockchains for cyber-physical systems

Matthew Edward Wagner

Follow this and additional works at: https://scholarsmine.mst.edu/doctoral_dissertations

 Part of the Computer Sciences Commons

Department: Computer Science

Recommended Citation

Wagner, Matthew Edward, "Secure blockchains for cyber-physical systems" (2020). *Doctoral Dissertations*. 2924.
https://scholarsmine.mst.edu/doctoral_dissertations/2924

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

SECURE BLOCKCHAINS FOR CYBER-PHYSICAL SYSTEMS

by

MATTHEW EDWARD WAGNER

A DISSERTATION

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2020

Approved by

Bruce McMillin, Advisor
Sanjay Madria
George Markowsky
Siddhardh Nadendla
Jonathan Kimball

Copyright 2020

MATTHEW EDWARD WAGNER

All Rights Reserved

ABSTRACT

Blockchains are a data structure used to perform state agreement in a distributed system across an entire network. One unique trait of blockchains is the lack of a centralized trusted third-party to control the system. This prevents a corrupted trusted third party from being able to control the entire blockchain. All nodes can reach agreement in an untrusted network where nodes do not need to trust one another to believe the accuracy of the information stored. Two main issues occur when trying to apply this technology to other applications: verifiability and scalability. In previous blockchain architectures, there is no way to validate off-chain data i.e. sensor reading. Some have purposed the use of a trusted third-party. Unfortunately, using a trusted third-party undoes a main advantage of blockchains and allows corruption to become a concerning possibility. Other challenges to applying blockchains to cyber-physical systems include keeping a single ledger up-to-date in real-time. The drawbacks of Bitcoin, a popular application of blockchains, have been very well documented in terms of speed.

The main purpose of this work is to address the verifiability and scalability issues of blockchains for cyber-physical systems. It proposes a solution that expands the application of blockchains to cyber-physical systems while maintaining the benefits. If the use of blockchains is to be expanded to off-chain data, they need to have the capability to securely encapsulate the physical world in a verifiable way. The following is a list of major contributions by the work: 1) propose a framework for verifying physical transactions in a blockchain, 2) propose a method to increase scalability and allow the use of blockchains in a disconnected network, 3) propose a truncation mechanism for cyber-physical transactions that allow for real-time speed. With these three contributions, this work introduces some additional ideas to blockchains and expands their applications.

ACKNOWLEDGMENTS

I would like to thank Aaryn for sticking with me through my graduate degree. Things haven't always been easy for us during my studies, but in the end, everything worked out. Without your support, I am not sure things would have gone as well as they did. Thanks for being my rock.

I would like to thank Professor Bruce McMillin for recruiting me to Missouri University of Science and Technology and being my advisor. He allowed me great freedom to find a topic that was truly interesting to me and gave me space and opportunities to perform research.

I would like to thank Coach Grooms for his mentorship and friendship during my time at Missouri S&T.

I would like to thank all the friends I made in Rolla. Whether in the lab, in the pool, or just in the classroom you all helped me enjoy my time at Missouri S&T.

This work was supported in part by the Missouri University of Science and Technology's Chancellor's Distinguished Fellowship and grants from the US National Science Foundation under awards CNS-1505610 and CNS-183747. That support was greatly appreciated and allowed me the freedom to pursue topics of interest.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

NOMENCLATURE

Symbol	Description
BGP	Border Gateway Protocol
BTC	Bitcoin
CA	Certificate Authority
CPS	Cyber-Physical System
CPU	Central Processing Unit
CRL	Certificate Revocation List
DFS	Depth-first Search
DOS	Denial of Service
GPU	Graphics Processing Unit
IRC	Inter-Roadside Communication
IVC	Inter-Vehicle Communication
LTL	Linear Temporal Logic
MSDND	Multi Security Domain Non-Deductiblility
PKI	Public Key Infrastructure
PoB	Proof-of-Burn
PoE	Proof-of-Event
PoS	Proof-of-Stake

PoW	Proof-of-Work
RSU	Road-Side Unit
USD	U.S. Dollars
USDOT	United States Department of Transportation
VANET	Vehicular Ad-hoc Network
VRC	Vehicle-to-Roadside Communication

1. INTRODUCTION

Since the roll-out of Bitcoin, blockchains have seen attention in the national limelight (Nakamoto *et al.*, 2008). This technology has been touted as revolutionary. It creates a ledger that is immutable where users do not have to trust any other users but can still reach an agreement. In a blockchain, everything is verifiable and users cannot lie about the data they have published. All of these attributes make blockchains an attractive technology to use to increase the resiliency of different systems. Many authors have attempted to apply blockchains to a wide variety of different applications including banking, land management, critical infrastructures, and many more.

Critical infrastructures include telecommunications, electrical power systems, gas and oil systems, banking and finance, water supply systems, emergency services, and transportation systems (Moteff *et al.*, 2003). These systems are essential for everyday living. In recent years, they have become targets for cyber-attacks (Wagner and Schweitzer, 2016). Currently, these attacks have been limited to systems with large cyber-component or central control systems such as the supervisory control and data acquisition systems that manage electrical power systems. However, it is only a matter of time before these attacks expand to other critical infrastructures including those proposed to manage national transportation systems.

With autonomous vehicles on the rise, national transportation systems are primed to become a key target for cyber-attackers. This becomes increasingly true with the large push for vehicular ad-hoc networks which help increase efficiency in our national transportation system resulting in large cost savings. However, many attempts to apply this technology to cyber-physical systems have largely failed to maintain the key principles of blockchains.

This dissertation proposes a solution to maintain the verifiability of off-chain data that is generated by cyber-physical systems, in particular vehicular ad-hoc networks. Off-chain data is data generated outside of the blockchain environment but stored within the blockchain. These can be sensor readings, transactions, and more. Since the data is generated off-chain, Bitcoin and other blockchains do not propose a way to verify this data. If the data in a blockchain is not verified, users must implicitly trust each other on the correctness, removing a key tenet of the technology.

This dissertation is organized as follows. Section 2 presents background information on technologies mentioned in this work including cyber-physical systems, vehicular ad-hoc networks, blockchains, digital signatures, and multi-security domain non-deductibility. The specific problems solved by this work are defined in Section 3. A literature review of related work is found in Section 4. Two schemes are proposed in Section 5 and Section 6 that maintain the verifiability of off-chain data for VANETs. These are the main contributions of this dissertation.

Section 5 includes a formal proof of the proposed communication protocol. Linear temporal logic (LTL) invariants are used in conjunction with a model of the proposed protocols created using the SPIN model checker to formally verify the scheme. This verification is used to show that the cyber portion of the proposed system behaves correctly.

Section 6 includes proofs using MSDND and formal logic to strengthen the argument that the proposed protocols behave correctly. These proofs show that the inclusion of physical attestation at the individual vehicle level allows for the vehicles to determine if other vehicles are behaving correctly. Thus, the input into the communication protocols is correct. The proofs contained in both sections work together to show that these proposed protocols generate a formally verified blockchain containing off-chain data. a key contribution of this work.

Additionally, Section 6 proposes some alteration to the communication protocols that solve the issues encountered from the scheme presented in Section 5. The conclusion and possible future research areas can be found in Section 7.

2. BACKGROUND

This dissertation focuses on addressing issues that occur when blockchains are extended to applications that involve the physical world. These include applications that involve the storage of information generated outside of the blockchain. In this section, some background knowledge on topics used in this work is presented. The main areas that are used and discussed throughout this dissertation are cyber-physical systems (CPSs), vehicular ad-hoc networks (VANETs), blockchains, digital signatures, and multi security domain non-deductiblility (MSDND).

2.1. CPS

CPSs are the integration of embedded computers with physical processes (Lee, 2008). These systems involve both cyber and physical components and information flow between them. There are a variety of different CPSs from airplanes to smart thermostats. However, the primary CPS discussed in this paper is the autonomous vehicle architecture. An autonomous vehicle can be viewed as a CPS due to both the driving software and the physical movements of the car. The software takes sensor readings from the physical world as input. It then uses these sensor readings to determine what actions it needs to take. The software then conveys these commands to devices that control the physical actions of the vehicles, i.e. the brakes, accelerator, driving wheel, etc. This, in turn, creates a feedback loop affecting the sensor readings.

CPSs are unique due to the increased attack and defense vectors present. In a CPS, an attack can occur on either the physical or cyber part of the system. In an autonomous vehicle, either the software or the sensors can be manipulated. If the sensors are forced to give incorrect information, then the software can be tricked to thinking something different is

happening. Thus, the software can be tricked to perform a potentially dangerous maneuver. Likewise, the software can be attacked directly and forced to perform a potentially dangerous maneuver. However, additional defense mechanisms can be put into place.

For CPS, the additional defense mechanism is physical attestation. Physical attestation is the concept of using the physical properties of a system to determine the ground truth (Roth and McMillin, 2013). In autonomous vehicles, physical attestation can be used by evaluating equations for velocity and distance to determine if sensors are behaving correctly. The idea is that if an attack is occurring on a sensor, i.e. the sensor for distance traveled, then a vehicle can use both speed and time to detect this attack. The main concept of physical attestation is to increase the difficulty of an attack on a CPS to occur. If an attacker can fabricate multiple sensor readings, then they are still able to accomplish their goal. One major drawback of physical attestation is that it is not easily transferable between systems. Every application has a unique physical system and possible constraints. Thus, much work must be completed for every CPS.

2.2. VANETS

A VANET is a system where vehicles set up a local ad-hoc network and communicate with one another for a specific purpose (Zeadally *et al.*, 2012) (Elsadig and Fadlalla, 2016) (Hartenstein and Laberteaux, 2008). It can be seen as a natural evolution of mobile ad-hoc networks where the mobile nodes are vehicles and have low resources while stationary nodes are infrastructure components and have high resources. Many different applications for VANETs have been proposed in the literature. The purpose of these applications ranges from entertainment and highway system efficiency to driving specific applications. The application of a particular VANET affects what its requirements are. The main type of applications this paper centers around is driving-based applications. Two main examples are pre-crash alerting and roadway efficiency (Hartenstein and Laberteaux, 2008). Pre-

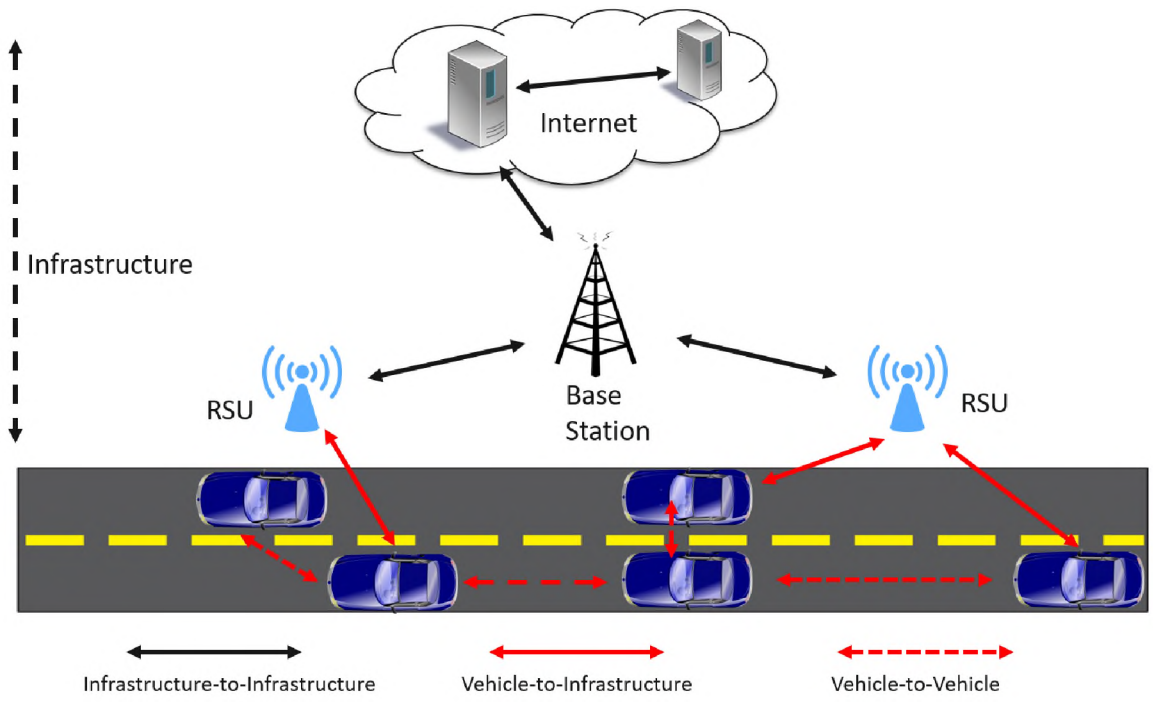


Figure 2.1. VANET Basic Architecture

crash alerting is when a vehicle alerts others that a crash has occurred behind them on the road so that they have increased time to reduce their speed and avoid an accident. Roadway efficiency involves cars working together to perform merging and tailgating maneuvers.

Currently, VANETs consist of basic architecture and several different communication paths as seen in Figure 2.1. In this network, there are several entities: vehicles, road-side units (RSUs), base stations, and any infrastructure entities that are needed for the given application. These infrastructure entities can include a certificate authority (CA), cloud servers and more to store passwords, user information, etc. The purpose of RSUs is to act as a component that has more computational resources and can act as a third-party between vehicles. Base stations act as a fast communication path between RSUs and the online entities. In the architecture, RSUs are built along all of the roadways. They can communicate with the base stations and any other infrastructure entities quickly due to the assumption that there is a wired connection running between them.

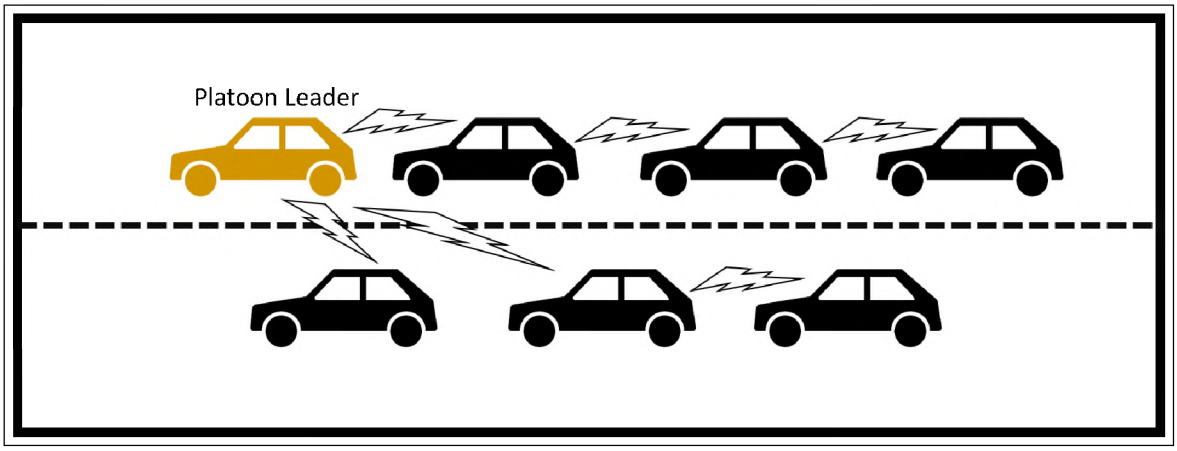


Figure 2.2. Platoon Example

One major drawback of the traditional architecture for VANETs is that RSUs are costly to set-up everywhere vehicles could be present. This dissertation considers an alternate architecture where no RSUs exist and there is minimum interaction with any infrastructure components. This model is more consistent with the traditional mobile ad-hoc network model (Goyal *et al.*, 2011).

In VANETs there are three main communication paths: inter-vehicle communication (IVC), vehicle-to-roadside communication (VRC), and inter-roadside communication (IRC) (Zeadally *et al.*, 2012). IVC occurs within a platoon when vehicles interact for a particular application. A platoon consists of a leader and members that are vehicles located closely together and moving in the same direction on a roadway. In a platoon, the leader issues commands to the rest of the platoon which are followed (Jia *et al.*, 2015). An example of a platoon can be seen in Figure 2.2. VRC occurs when vehicles communicate with RSUs. IVC occurs when RSUs communicate with one another or with any additional infrastructure components such as a user information database. However, this work primarily uses IVC due to the assumption that setting up RSUs would be too costly in practice.

As of now, many different solutions have been presented to implement security in VANETs. Most proposed solutions rely on RSUs and other infrastructure components and present some variations of a public key infrastructure (PKI) (Zeadally *et al.*, 2012). A PKI is a system used to distribute, use, store, and revoke digital certificates. These systems primarily focus on applying public-key encryption.

One major issue with using PKIs is the required infrastructure and trusted third-party needs to be consistently communicating with the nodes in the system. The CA is in charge of managing the users in the system by certifying and revoking users. A list of revoked users is stored in a Certificate Revocation List (CRL). This list must be updated for all participants every time a revocation occurs. As previously stated, the main issue with this approach is communication with the RSUs, which are costly to create. Thus, without the RSUs and the other infrastructure entities, the whole system struggles to remain updated when new vehicles are removed since this information must be broadcast network-wide. So, a PKI is not practical due to this cost.

To substantiate this claim, a rough cost estimate for an RSU infrastructure for the United States of America is calculated to show the motivation for its removal in the proposed work. In (Wright *et al.*, 2014), they built sample RSUs at multiple locations and calculated the full costs for creation and maintenance. The estimated cost to set-up an RSU is $51,600 each while the upkeep is between $1,950 and $3,050 per year. The study also states that it takes approximately fifty-five RSUs to cover four square miles. The United States consists of 3,535,948.12 square miles of land between all fifty states and the District of Columbia (Bureau, 2009). To calculate the cost of creating and maintaining the infrastructure for a VANET, the number of RSUs required is needed first. This is done by dividing the total square miles of the United States by four and multiplying it by fifty-five as per the study.

$$\frac{3,535,948.12 * 55}{4} = \$48,619,286.65 \tag{2.1}$$

This number is rounded to the nearest integer since you can not have half an RSU. To figure out the total installation cost, the number of RSUs is multiplied by the cost per RSU to get:

$$48,619,287 * \$51,600 = \$2,508,755,191,140 \tag{2.2}$$

Lastly, to calculate the maintenance cost of the VANET, multiply the number of RSUs by the annual estimated maintenance cost.

$$48,619,287 * \$2,500 = \$121,548,217,500 \tag{2.3}$$

With an initial cost of $2.5 trillion, a VANET architecture would be impractical to deploy. On top of the start-up cost, the yearly maintenance cost for an RSU infrastructure is approximately $121.5 billion per year which does not include the cost to maintain the backend servers, software, etc. These estimates are an upper bound on the cost for deploying RSUs since there are large swaths of land that have no roadways and would not require RSUs. However, there has been work that shows that a full network of RSUs that covers all roadways and is interconnected via land-line results in a tremendous improvement in terms of connectivity and message dissemination compared to the modest improvements by a sparse set of disconnected RSUs (Reis *et al.*, 2013). Thus, the RSU infrastructure needs to cover a significant portion of the total square miles of the United States for a VANET to be deployed universally.

With 2.3 million crashes a year, the proposed cost-saving of implementing a VANET system is $202 billion in costs in crash prevention alone (Wright *et al.*, 2014). This does not include savings due to decreased emissions or fuel consumption which would surely decrease due to other proposed VANET applications. Even though this could save the United States $80.5 billion per year, it would take over 31 years before it reached a point where they were financially benefitting from the system. Regardless of the cost savings, the study

mentions that the majority of the funding for such a roadway system would have to come from states and third-parties that want to create VANET applications. They specifically mention that the overall funding would be difficult to raise at the time of the study. However, the need for this VANET system and related applications is real. The prohibitive cost of this system and the complex process of funding it serves as motivation for this work and the proposed architecture which removes RSUs.

2.3. BLOCKCHAINS

Blockchains have seen rising interest in different fields since its creation in 2008 by Satoshi Nakamoto. This technology was first used with the creation of Bitcoin (BTC). It has since seen expanded use as a state agreement mechanism and a computationally immutable ledger in a wide variety of different cryptocurrencies and other applications. This technology consists of a few major components: transactions, scripts, a hash chain, a consensus mechanism, and a decentralized network. These parts are described in more detail for the BTC blockchain.

2.3.1. Transactions. The blockchain consists of a ledger of transactions. Every transaction consists of a set of inputs, a set of outputs, a digital signature, and a script that is used to validate the transaction. Transactions can be changed to contain any data as long as they are secured by a digital signature. The digital signature allows users to verify the transactions and support non-repudiation as a property in the system. In BTC, the transactions contain a list of digital addresses as input and output. The amount of coin being transferred too and from each of the accounts is included.

2.3.2. Scripts. For every input and output for a transaction, there is a script that is attached to the input or output (Tschorsch and Scheuermann, 2016) (Bonneau *et al.*, 2015). These scripts are an essential yet powerful building block for blockchains that let users understand how to interrupt and verify transactions. In BTC, they implement a non-Turing complete stack-based language. There is a common script that is used for most transactions

called the Pay-to-PubKeyHash. However, depending on the mining node other types of scripts can be accepted and published. The Pay-to-PubKeyHash is made of up of *scriptSig* and a *scriptPubKey*. The *scriptSig* is the input arguments for the script. It consists of sigX and pubKeyX or the signature and public key of person X who is sending the currency. The *scriptPubKey* is the set of operations that are to be performed to verify and unlock the transaction. For the Pay-to-PubKeyHash the *scriptPubKey* is OP_DUP, OP_HASH160, pubKeyHash, OP_EQUALVERIFY, OP_CHECKSIG. OP_DUP duplicates the most recent entry on the stack. The OP_HASH160 hashes the most recent entry twice, which should generate the address of the receiver, and pushes it onto the stack. The pubKeyHash is simply the receiver's address on the stack. The OP_EQUALVERIFY verifies the equality of the two topmost stack entries and raises an error if they differ. Lastly, OP_CHECKSIG checks the input signature against the public key and pushes true onto the stack if they match. With this building block users can create a wide variety of different scripts such as *m-of-n* multi-signature transactions which require *m* valid out of *n* possible signatures to redeem a transaction. However, other cryptocurrencies such as Ethereum provide a much greater variety of scripts that can be used for smart contracts (Christidis and Devetsikiotis, 2016).

2.3.3. Hash Chain. In BTC, the hash of the previous block is included in the current block. This creates a chain of hashes that signify the order of events and implement a logical timestamp server (Nakamoto *et al.*, 2008). This allows the system to maintain synchronization across the entire network without a trusted third-party.

2.3.4. Consensus Mechanism. This is the process used to create new blocks and reach consensus. These blocks represent the agreed state of the system. In BTC, Proof-of-Work (PoW) is used. To publish a new block using PoW, a user increments a nonce in the block until the block's hash is less than some predefined number. The time it takes to complete this process increases exponentially as the difficulty of the problem increases. However, the result can be calculated in constant time by executing a single hash. Thus, the outcome is that any change in the previous blocks requires the change of the hash of

that block and any blocks that come after it, creating a computationally immutable order of events. Since a range of answers is allowed and all nodes in the network are racing to find it, multiple nodes can produce a correct answer at the same time, results in a phenomenon called forking. Forking is where multiple correct blocks were created at the same time and forwarded across the network, creating a conflict in the order of events.

2.3.5. Decentralized Network. The BTC network consists of a large network of nodes that are all competing to create the next block to receive a reward. Nodes are allowed to join and leave any time. There are several policies in place to resolve issues from the concurrent nature of the system since every node should eventually have the same blockchain to verify new transactions. In BTC, the longest hash chain takes over as the agreed-upon series of events. This policy helps prevent forking, which is the event of multiple blocks being generated at the same time due to the range of different acceptable solutions that can be generated during the PoW process.

In Bitcoin, transactions are used to transfer currency between accounts. This currency is generated by using your computing power to find the correct PoW which leads to a publishable block of transactions. These blocks are then chained together by including the hash of the previous block in this current block. This creates a timeline of transactions that are used to avoid users sending a coin to two different addresses, called the double-spending issue.

2.3.6. Properties of Blockchains. Blockchain has three fundamental attributes that should be present in any other application of the technology: immutability, verifiability, and non-repudiation. Immutability is upheld in Bitcoin through the use of PoW. In general, Bitcoin is considered immutable due to the computational power it would require to regenerate every block after a change is made in a previous block due to the PoW mechanism. However, it is only computationally immutable since there is a possibility someone with enough computation power could outpace the entire Bitcoin network and determine the order of transactions as desired.

Verifiability can be broken down into two different areas: the order of events and the data. In Bitcoin, the order of events is ensured via the hash chain and the immutability property listed before. Any user can check that the order of events is correct by simply verifying the entire hash chain up until the current block. Data is verified via calculating the current balance of a sender's account and checking that it is greater than or equal to the amount they are sending to someone else.

Non-repudiation means that users cannot lie about what they entered into the blockchain. This is guaranteed through the use of digital signatures. Once a transaction is stored using that user's digital signature, they will be unable to deny that they sent the transaction since only they should have access to the keys required to make the signature.

2.3.7. Smart Contracts. As mentioned previously, one major development that has occurred since the invention of blockchains has been smart contracts. Smart contracts are scripts that are executable by the blockchain. The idea of smart contracts has been heavily implemented and developed on a cryptocurrency named Ethereum. The suggested use of smart contracts has been for deploying any real-world contract on a blockchain. This can allow for business dealings, homeownership, and even gambling to occur on a blockchain (Buterin *et al.*, 2014).

Currently, many different applications of blockchains have been proposed. These include digital identity providers, online voting systems, decentralized storage, public notary, managing music royalties, document proof-of-existence, distributed Domain Name Server, distributed PKI, and many more (Pilkington, 2016)(Crosby *et al.*, 2016). All of these applications share one common requirement: a distributed, untrusted networked without a trust centralized authority. Without these attributes, no applications need to use a blockchain and will benefit from storing information in traditional ways such as a database.

2.4. DIGITAL SIGNATURES

In the proposed scheme, both asymmetric digital signatures and multi-digital signatures schemes are used. An asymmetric digital signature scheme is a scheme where there exist some public and private key owned by a signer. The signer can sign a message using their private key that can be verified using their public key. The proposed protocols do not specify an asymmetric digital signature scheme. Thus any scheme, such as the Elliptical Curve Digital Signature Algorithm, can be used (Johnson *et al.*, 2001).

To generalize how these schemes work, the digital signature scheme has some key generation function, signing function, and verification function. The key generation function is: $(Key_{Public}, Key_{Private}) \leftarrow KeyGen(X)$ where Key_{Public} is the public key, $Key_{Private}$ is the private key, and $KeyGen(X)$ is the key generation algorithm given some set of inputs X. The signing function is $M' \leftarrow Sign(Key_{Private}, M)$ where M' is the signed messaged and $Sign(Key_{Private}, M)$ is the signature creation function given an input private key $Key_{Private}$ and a message M. The verification function is $Output \leftarrow Verify(Key_{Public}, M')$ where $Output$ is either accept or reject and the verification algorithm $Verify(Key_{Public}, M')$ with and input key Key_{Public} and signed message M'.

A multi-digital signature scheme is a protocol that allows a group of signers to produce a short, joint signature on some common message. This message can be verified using the group public key that is generated when signing the message (Maxwell *et al.*, 2018). This paper uses Schnorr multi-signature. An adaptation of the scheme is presented in section 6.1 as part of the proposed protocols.

In (Maxwell *et al.*, 2018), there are three steps: key generation, signing, and verification. This paper presents the scheme adapted to our model. To see the original scheme, please refer to the original paper.

2.4.1. Key Generation. Each signer generates a random private key $x \leftarrow \mathbb{Z}_p$. Each signer computes the corresponding public key $X = g^x$.

2.4.2. Signing. Let X_1 and x_1 be the public and private key of a specific vehicle, let m be the message to sign, let $X_2, ..., X_n$ be the public keys of all the other vehicles in the platoon, and let $L = X_1, ...X_n$ be the multi-set of all public keys involved in the block creation process.

For $i \in 1, ..., n$, each vehicle computes $a_i = H_{agg}(L, X_i)$. The aggregated public key for the platoon is $\tilde{X} = \prod_{i=1}^{n} X_i^{a_1}$.

Next, each platoon member generates a random $r_1 \leftarrow \mathbb{Z}_p$, computes $R_1 = g^{r_1}$, and $t_1 = H_{com}(R_1)$.

Each platoon member sends t_1 to all other members of the platoon.

Once it gets $t_2, ..., t_n$ from the other platoon members, it sends R_1. Once it gets $R_2, ..., R_n$ it checks that $t_i = H_{com}(R_i)$ for all $i \in 2, ..., n$.

If not true abort, else compute $R = \prod_{i=1}^{n} R_i$, $c = H_{sig}(\tilde{X}, R, m)$,, $s_1 = r_1 + ca_1x_1 modp$.

The vehicle sends s_1 to all other platoon members. Once it gets $s_2, ...s_n$ from platoon members, it computes $s = \sum_{i=1}^{n} s_1 modp$ and the signature is $\sigma = (R, s)$.

2.4.3. Verification. Given a multi-set of public keys $L = X_1, ..., X_n$, a message m, and a signature $\sigma = (R, s)$, the new platoon compute $a_i = H_{agg}(L, X_i)$ for each $i \in 1, ..., n$, $\tilde{X} = \prod_{i=1}^{n} X_i^{a_i}$, $c = H_{sig}(\tilde{X}, R, m)$.

The new platoon accepts the certification if $G^s = R \prod_{i=1}^{n} X_i^{a_i c} = RX^c$.

2.5. MSDND

MSDND is a method created for evaluating the information flow across in a system to formally analyze the trust in cyber-physical systems using modal logic (Howser and McMillin, 2014). The formal definition of MSDND is given below.

$$\text{MSDND(ES): } \exists w \in W \vdash [(s_x \vee s_y)] \wedge \sim (s_x \wedge s_y)$$
$$\wedge [w \models (\nexists V_x^i(w) \wedge \nexists V_y^i(w))]$$

This can be simplified to the following definition based on basic Boolean logic and the definition of exclusive-or.

$$\text{MSDND(ES): } \exists w \in W \vdash [(s_x \oplus s_y)]$$
$$\wedge [w \models (\nexists V_x^i(w) \wedge \nexists V_y^i(w))]$$

In the expression, $V_x^i(w)$ and $V_y^i(w))$ refer to valuation functions on the state variables s_x and s_y accordingly. It is important to note that if a system or information flow path is MSDND secure, then it is vulnerable to a Stuxnet-like attack in a model that is trying to maintain high integrity. However, if it is MSDND secure, then the architecture is secure under a privacy model.

The proofs presented in this dissertation use a shorthand for MSDND proofs created in (Palaniswamy and McMillin, 2018). The theorem $IBT_{1,2}Val$ states that entity two sent information to entity one, entity one believes that Val is true, entity one trusts the information sent by entity two, and that entity one believes Val is correct. This command is used to show information flow across an information flow model.

MSDND proofs are presented in Section 6.2 to show the security of the proposed schemes. They are used to show that a cyber-only or physical-only blockchain is insecure in the face of these attacks and that the cyber-physical blockchain presented in the paper is secure to the same attacks.

3. PROBLEM STATEMENT

Securing VANETs using blockchains, in contrast to using it for a cryptocurrency, raises five main problems. These problems include cyber-only transactions, a system-wide ledger in a disconnected network, real-time requirements for transactions, and the lack of registration for participants and are discussed in more detail in the following subsections. These problems stem from issues with verifiability and scalability when adapting blockchains to other applications. The solution proposed in this dissertation addresses all of these issues.

3.1. CYBER-ONLY TRANSACTIONS

In BTC, several mechanisms exist to verify that the cyber transactions have occurred and that the receiving party is not cheated. These mechanisms ensure properties such as verifiability and non-repudiation. In the current architecture adopted by BTC and other cryptocurrencies, any off-chain aspect of the transaction is not guaranteed.

Let's take the example of Alice trading BTCs to Bob in exchange for U.S. Dollars (USD). In this example, Alice could send the BTC to Bob, Bob could verify that the BTC has been received, then refuse to send Alice the USD. Similarly, Bob could send Alice the USD, and Alice could refuse to pay Bob the BTC once she receives the USD. The off-chain aspects of these transactions aren't managed by the blockchain. Instead, the users must rely on a trusted third party to ensure the transaction has taken place.

The lack of verification mechanisms for any off-chain aspects of the transactions becomes a significant issue when trying to apply blockchains to CPS. For example, in a VANET, vehicles need to be able to physically verify traffic or driving messages from other vehicles. The ground truth in a CPS is that actions occurring in the physical domain of the system. Blockchains require some way to verify off-chain information and verify the

ground truth. Additionally, in a platoon setting, vehicles need to be able to verify that other vehicles followed instructions. Attempts have been made to apply blockchains to other CPS. These attempts are further discussed in Section 4. The inability to verify any off-chain data contained in transactions and determine the ground truth of CPSs is a major issue with blockchains. However, it is solved in this dissertation by leveraging platoons in a VANET to capture the ground truth of local vehicles.

3.2. A SYSTEM-WIDE LEDGER IN A DISCONNECTED NETWORK

This dissertation focuses on a VANET system leveraging platoons with little reliance on RSUs and other infrastructure components. The overall architecture of a VANET can be viewed as a disconnected graph where partitions of nodes have no connection with others outside their partition for long, perhaps indefinite, periods. Vehicles can easily connect with other vehicles nearby or those within their platoon but they may have difficulty connecting with vehicles in different cities. This architecture proposed a major challenge to blockchains as compared to BTC where every node is required to eventually maintain the same blockchain since future transactions must be verified using previous transactions.

In a VANET simply maintaining the blockchain across the entire network would become an issue without a heavy reliance on an expensive RSU infrastructure. A solution involving local consensus is proposed to solve this issue.

3.3. REAL-TIME TRANSACTION REQUIRMENTS

The transaction throughput of Bitcoin is between 3.3 and 7 transactions per second with an average block creation time of 10 minutes (Kiayias and Panagiotakos, 2015). This is due to the difficulty of the PoW hash puzzle. In VANETS, transactions need to be processed quickly, in real-time due to the speed of the physical driving actions.

For example, assume every vehicle in a platoon generates a transaction for every other vehicle every time a maneuver is performed. This would create hundreds of transactions every minute. Additionally, malicious or malfunctioning vehicles need to be revoked promptly to avoid accidents and save lives. Thus, by applying BTC architecture, logging these transactions with physical-based time restraints are impractical due to the increased throughput and overall speed requirements. In its current form, blockchains struggle to achieve the speed required by vehicular driving applications. This dissertation proposes a different consensus mechanism to address this issue.

3.4. NO REGISTRATION FOR PARTICIPANTS

Nodes in BTC are allowed to join and leave the network at any time. There is no CA which registers or keeps track of users. Instead, a peer-to-peer system is used to maintain the network. This occurs due to the nature of the architecture and the fact that there are no negatives to new users participating in the system as long as they follow the rules. Users are told to create a new account for every transaction to increase their privacy. This increases the throughput of nodes joining and leaving the system.

There are many negative impacts of a public blockchain when applied to a CPS. Malicious nodes could claim to be multiple different users and perform a Sybil attack (Elsadig and Fadlalla, 2016). A Sybil attack is when a user pretended to be multiple entities at once. Additionally, in a VANET there is an extensive number of laws and safety regulations to keep users safe in the physical system. In these systems, there is a great need for a CA to verify that physically the nodes are following the rules initially and at various time intervals. A CA is introduced in the proposed solution to solve this problem.

3.5. 51% ATTACK IN POW

The 51% attack occurs when one node or a group of nodes works together to have 51% of the network's mining hash rate, hence the name. This attack can only occur in blockchains that use the PoW or equivalent consensus protocols. With the majority of the hash rate, the miner or group of miners can control any new blocks that get created. This means they can decide what transactions go through. With this power, they can require a specific transaction fee or refuse to publish a transaction. Thus, wealth will be generated through the collection of fees on top of their ability to decide who can do what. They can also revert previous blocks and cause double-spending of coins by publishing a block then republishing different transactions. This attack allows the attacker to become the sole governor of the entire blockchain.

The 51% attack is long thought of as being impossible to perform on BTC due to the high computational requirements required. For this reason, there have been many suggested applications from maintaining currencies for a nation to managing land titles and property deeds via smart contracts. This is especially important for developing countries where corruption is often rampant. So, let us take an example where a country implemented a BTC-based blockchain to maintain a CPS required to maintain a national infrastructure. A different country may want to attack this system as a means of crippling that country.

In the past, there have been a wide range of cryptocurrencies smaller than BTC that have been attacked in this way. This includes Ethereum, the 2nd largest cryptocurrency in terms of market capitalization at $26,089,510,029 as of 06/06/2019. The most recent one of these attacks occurred in January of 2019 (Rodgers, 2019). This is easier for smaller cryptocurrencies because the attack only needs 51% of the total hash rate of the network. This value is based on the number of people participating in the PoW process and adding to the total hash rate. Thus, the more people competing to gain the reward from creating the block, the harder it is to get a 51% attack. This is why many people still believe BTC to be safe due to the number of users participating. To put it in perspective, the highest recorded

daily average hash rate for Bitcoin Cash was 8.3299 EH/s (quintillion hashes per second) while Bitcoin's was 61.866256 EH/s. Bitcoin Cash is the 4th largest cryptocurrency in terms of market capitalization. BTC's hash rate has 750% of the amount of Bitcoin Cash's hash rate.

To participate in PoW mining, a node needs to be able to solve the mathematically hard problem and generate hashes. In terms of hardware, graphics processing units (GPUs) are better for mining BTC than central processing units (CPUs). CPUs can be thought of as the manager of your computer while GPUs are the laborer. Likewise, dedicated mining hardware, like Bitmain's Antminers, is better than GPUs. These devices are specifically designed to generate hashes. They often have just enough memory to perform the computations required. CPUs are measured on H/s (hashes per second), GPUs are measured on MH/s (million hashes per second), and dedicated mining hardware is measured on TH/s(trillion hashes per second). This means that supercomputers such as the Summit and Sierra in the United States don't have the computation power to perform this attack on Bitcoin since they only have GPUs and CPUs.

However, dedicated mining hardware must be specifically designed for different mathematical problems. When the mathematical problem changes, these devices don't work. For example, Ethereum uses the Ethash hash function for its mathematical problem. To solve this problem, significantly more memory is required than the problem used to mine BTC. Thus, Bitmain's Antminers do not work for this cryptocurrency. In this case, GPUs are the preferred hardware for mining since they have the appropriate amount of memory. This is also the case for other hash algorithms that are used.

The best hardware option for the algorithms used for some of the top ten biggest cryptocurrencies (CoinMarketCap, 2019) was found to determine the cost of the attack. There is currently no standard for calculating the hash rate of the hardware. Instead, users have participated in mining and reported the speeds. Three cost-effective hardware solution for three different hash algorithms was gathered. Table 3.1 shows the gathered information.

Table 3.1. Mining Hardware

Name	Type	Hash Function	Speed	Cost
Antminer S9j (Alibaba.com, 2019)	Dedicated	SHA-256	14.5 TH/s	$379.00
Radeon V11 (Hanson and Uy, 2020)	GPU	Ethash	100 MH/s	$680.00
Antminer L3++ (Value, 2019)	Dedicated	Scrypt	596 MH/s	$379.98

The first step to calculate the cost of the 51% attack is determining how much hash rate the attacker needs to have. Since they will need a majority after the attack starts, it can be assumed they will be adding 51% additional hash rate to the network. The hash rates for the cryptocurrencies we will be comparing can be seen in Table 3.2 (BitInfoCharts, 2019a,b,c; Blockchain.com, 2019; BuriedOne, 2019). To determine the required hash rate for the attack, we use the following equation.

$$requiredhashrate = \frac{highesthashrate}{.49} - highesthashrate$$

Next, the number of hardware components required to get to that hash rate needs to be calculated (rounded up to the nearest device).

$$numberofdevices = \frac{requiredhashrate}{hashrateofdevice}$$

Lastly, the cost of that many hardware components is calculated.

$$costofattack = numberofdevicesxcostofthedevice$$

Even though the cost of the hardware component alone does not encompass the complete cost of the hardware nor the cost of running the attack, it is a good metric to determine the feasibility of the attack. As seen in Table 3.2, it only costs $1,683,058,273 in hardware to perform the 51% attack on BTC. With a cost of under two billion dollars, most countries could afford to perform this attack. Many private corporations could also do this. The table also confirms the belief that smaller cryptocurrencies are even cheaper to attack. This is significant for any new blockchains that may be created. They will be more vulnerable until they gain a significant pool of users contributing to mining.

Overall, this shows the serious threat a 51% attack has on BTC-based blockchains. Using the same architecture for anything highly-valuable puts those things at risk.

Altering the consensus mechanism could make a blockchain completely resilient to the 51% attack by removing the race to solve the puzzle that the attack relies on. One such example is the Proof-of-Stake (PoS) consensus mechanism that is set to be adopted by the Ethereum cryptocurrency this year. PoS no longer relies on miners to solve mathematically hard problems to be able to create the next block. Instead, PoS attributes the number of blocks a user can mine to the amount of stake they have in the system (Bentov *et al.*, 2016). The philosophy here is that a user with more stake, a blockchain is more likely to behave appropriately since their stake could be at risk if they do not. This results in the more stake you have, the more blocks you can write, and the more power you have over others in the system. However, it opens itself up to different attacks, but one that may be impossible for another nation to pull off without buying a majority of the stake in the blockchain. This dissertation utilizes an alternative consensus mechanism to solve this problem.

Table 3.2. 51% Attack Cost

Name	Market Cap	Highest Hash Rate	Hash Function	# of Devices	51% Attack Cost
Bitcoin	$141,863,353,114	61.866256 EH/s	SHA-256	4,440,787	$1,683,058,273
Ethereum	$26,634,147,840	295,911.9974 GH/s	Ethash	3,079,901	$2,094,336,760
Bitcoin Cash	$7,049,828,808	8.3299 EH/s	SHA-256	597,924	$226,613,196
Litecoin	$6,520,122,343	370.5707 TH/s	Scrypt	647,141	$245,900,637.18
Bitcoin SV	$4,043,291,830	4.374 EH/s	SHA-256	313,968	$118,993,872

4. LITERATURE REVIEW

This section discusses previous work done in the area of VANETs and blockchains. In particular, past proposed security solutions for VANETs, proposed reputation systems for VANETs, the security of blockchains, proposed enhancements of blockchains, and proposed uses for blockchains are presented.

4.1. VANETS

A significant amount of research has been done on VANETs. Many proposed solutions attempt to secure VANETs in some aspect or another. Additionally, some reputation-based systems have been created. However, they tend to fail short when compared with the solution proposed in this work. This dissertation details how many solutions suffer from a reliance on infrastructure components when trying to secure VANETs.

4.1.1. Security for VANETs. Security is an important topic for any architecture. This is especially true in VANETs where malicious actions can cost the lives of the users. A variety of different papers have proposed solutions to solve different security concerns in VANETs. All of the solutions discussed here revolve around key management.

Key management involves managing digital signature and encryption keys and revoking users from the entire system if they misbehave or deemed malicious. Generally, this relies on a trust third party to manage all of this activity. The trusted third party is assumed to be good. This architecture is used in (Alexiou *et al.*, 2013; Kamat *et al.*, 2006; Singh *et al.*, 2015; Squicciarini *et al.*, 2011; Studer *et al.*, 2009; Walker, 2017; Xie *et al.*, 2016). All of these proposed schemes attempt to uphold some security attributes from authentication and privacy preservation in (Singh *et al.*, 2015) to authentication, authorization, and accountability in (Alexiou *et al.*, 2013). The main goal that ties these papers together is the goal of private communication, authentication, and accountability.

This means that these schemes want to keep communication between vehicles private, only allow a certain vehicle to communicate via forcing them to authenticate and hold vehicles accountable to what they say to other vehicles.

A major problem with all of these different solutions that want to perform key management is the reliance on a trusted third party. For example, (Singh *et al.*, 2015) suggests deploying RSUs to every intersection to handle the volume of messages across an entire city. Likewise, (Alexiou *et al.*, 2013) suggests the use of pseudonyms and a CRL. Even the United States Department of Transportation (USDOT) has created proposed a sample VANET architecture to serve as a PKI and manage a CRL. In all of these solutions maintaining a CRL would require massive resources to constantly monitoring of all traffic across the entire network and broadcast the information of revoked users. Some other solutions, such as (Kamat *et al.*, 2006), attempt to maintain privacy for vehicles identity-based cryptography for the same privacy. However, accountability is maintained by a trust third party in the form of RSUs who are the only ones who can link vehicles to their past transactions. Our proposed solution allows authentication, accountability, and private communication without any reliance on an infrastructure component such as an RSU outside of the initialization of the system. Without costly RSUs, all of these systems cannot operate.

4.1.2. Reputation Systems for VANETs. Another approach to securing VANETs is reputation-based systems. Rather than determining what information is correct or incorrect at any given time, reputation systems allow vehicles to learn what sources can be trusted while applying their ground truth to the given information. Multiple different reputation systems have been proposed for VANETs. All of these proposed schemes determine the ground truth of the system and determine whether a message was valid and thus the vehicle that sent it trustworthy. In (Gurung *et al.*, 2013), the scheme examines content similarity, content conflict, and route similarity between messages to determine the trustworthiness of the content of a message. Similarly, in (Jaimes *et al.*, 2016) the scheme relies on RSUs entirely to valid messages. The validation method most similar to ours is the one found

in (Chen *et al.*, 2013). In their work, other vehicles generate feedback messages about a vehicle's message. Thus, the system grades a vehicle's trustworthiness on the feedback from other vehicles. This method is better than both (Gurung *et al.*, 2013) and (Jaimes *et al.*, 2016) since the infrastructure alone has no way to learn the ground truth of the system. RSUs don't have sensors to evaluate a system and message similarity could be faked via many malicious vehicles reporting incorrect information. However, none of these validation methods from previous works take things far enough. In our solution, a vehicle's trustworthiness is graded based on all of its actions and messages, not just what they say they do. This is an important distinction since the vehicle can claim to be good and doing the correct thing while simultaneously behaving erroneously and maliciously.

All three of the previously proposed schemes rely heavily on infrastructure components to do the majority of the work. In (Gurung *et al.*, 2013) and (Jaimes *et al.*, 2016), the messages are compared and evaluated by an infrastructure component. In (Chen *et al.*, 2013) vehicles are evaluated by one-another but the feedback is aggregated and trust is rewarded by an infrastructure component. This heavy reliance on infrastructure components such as RSUs and evaluation servers is a major drawback compared with the approach proposed in this work. The approaches proposed in this dissertation do not require any third party outside of the initial set-up of the system. Thus, it would benefit from huge savings if it were to be deployed compared to these other approaches. Additionally, the proposed approach doesn't suffer from sparse RSUs and evaluations generated by the infrastructure components. In the previously proposed solutions, such sparse RSUs would allow vehicles to behave maliciously until their certification has been revoked. In the proposed solution, this isn't an issue since trust is generated and can be taken away by the platoon of vehicles itself.

Now that previous work done in VANETs has been discussed, relevant work done in blockchains is presented.

4.2. BLOCKCHAINS

The research area of blockchains has only been around since it was started in 2009. However, a great deal of work has already been done. In particular, many researchers have examined the security of the data structure while others have looked at ways to use it for new applications. However, this technology has thus far failed to encapsulate the physical world in a verifiable way. Some suggest leaving the verification of the physical world to a trusted third party. However, when a trusted third party is included, nodes have to take them at their word which opens the door to corruption by the trusted entity. Even if it is assumed that the trusted third party isn't lying, the nodes have no way to verify their information which counteracts the idea behind a trustless distributed network.

4.2.1. Security of Blockchains. Many researchers have examined a variety of different attacks on BTC and other blockchains. These can be broken into network-level attacks and attacks on and by mining pools.

4.2.1.1. Network level attacks. Network-level attacks focus on attacking the network of BTC and other cryptocurrencies. Currently, BTC uses standard networking procedures, such as the Border Gateway Protocol (BGP) and an anti-DOS protection protocol. BGP is the standard routing protocol that regulates how traffic is forwarded on the Internet. Within BGP users are allowed to claim packets that should go to a certain prefix. The BTC anti-DOS protection protocol implements a reputation-based system where each node keeps a penalty score for every other BTC peer. Whenever a malformed message is sent to the node the peer's penalty score is increased until it receives a twenty-four-hour timeout. Both of these systems have been shown to have weaknesses in BTC.

The BGP protocol was maliciously used to perform a partitioning in (Apostolaki *et al.*, 2017). The goal of the partitioning attack is to completely disconnect a set of nodes from the network. This essentially requires the attack to divert and cut all the connections between the set of nodes and the rest of the BTC network. The effect is that both sides of the BTC network believe their blockchain and their view is the correct one. So they continue

mining and increasing the length of their blockchain. However, once the two parts become connected again they realize that only one view is correct, essentially wasting the work of the other part whose blockchain doesn't become the main chain. (Apostolaki *et al.*, 2017) notes that this attack can become an interception attack if they allow one path to go through between the two sets of nodes. They found that a real BGP hijack performed against their nodes only takes two minutes to divert BTC traffic and that it would only take 39 prefixes to isolate 50% of the overall mining power in BTC.

BTC's native DOS protection scheme was also abused in (Apostolaki *et al.*, 2017; Biryukov and Pustogarov, 2015; Gervais *et al.*, 2015). All of these works show that DOS protection can be used to perform a delaying attack on the BTC network. The delay attack leverages the fact that GETDATA and BLOCK messages are not protected against tampering within the BTC network. Within the protocol, the GETDATA message is used to asks for blocks from other nodes while the BLOCK message is used to send a block to another node. Additionally, nodes are required to wait twenty minutes between requests to ask another node for a block. So, intercepting these messages and then corrupting them allows for a node to delay the propagation of a new block to another node by up to twenty minutes. This was shown for regular networks in (Apostolaki *et al.*, 2017; Gervais *et al.*, 2015) and specifically for the TOR network in (Biryukov and Pustogarov, 2015).

One major benefit of the solution proposed in this dissertation is that network-level attacks aren't an issue. In the proposed scheme, vehicles are assumed to be within a platoon. The proposed solution requires that vehicles be physically present within a platoon. This produces the assumption that all vehicles will be within a one-hop broadcast distance. Thus, there won't be any forwarding of messages, and all vehicles within a platoon will be able to communicate directly with one another. So, no partitioning or delaying attacks can be achieved since there won't be any forwarding of blocks.

4.2.1.2. Mining pool attacks. Within BTC and other cryptocurrencies, it is common practice for miners to work together to validate new blocks, forming mining pools. These pools work together to solve the proof of work puzzles and share the profits among themselves. Mining pools are contrary to the idea of a "One CPU, One Vote" system since multiple parties can pool their money to have control of the system. (Eyal, 2015) discussed the mining pool in-depth. The only time mining pools have been subject to criticism is when a mining pool GHash.IO reached the 50% mining power threshold in June 2014 which is the theoretical amount of computing power one user would need to have majority control of the blockchain. Since then no mining pools have attempted to increase their mining power to the system threshold.

(Eyal, 2015) presents a game theory approach to mining pools. In this game, mining pools have two options, attacking or agreeing not to attack. Attacking in the scenario refers to the block withholding attack where a miner sends only partial proof of works to the pool manager and discards full proofs of work. The partial proofs of work allow the miner to stay in the pool without actually helping mine new blocks. In return, they have also rewarded a portion of the reward. For this game, the authors found that the best long term goal is to agree not to attack while the best short term goal is to attack other mining pools. This is a similar scenario to the classical Prisoners Dilemma game. They found that currently, mining pools do not attack one another since it decreases their own earnings potential and reduces the overall mining potential of the whole system since there is an overall decrease in mining power. From their work, they determined that since most mining pools are open to the public there is relatively little way to protect against these attacks other than by monitoring the rates at which all the miners are giving the pool manager full proof of work. They suggest one workaround would be to require miners to pay an entry fee to join the pool.

However, mining pools and attacks on mining pools are not an issue for the proposed scheme. In the proposed scheme, the PoW algorithm is not used. No reward is given for creating new blocks or creating transactions. Instead, these operations are a requirement to participate in a platoon and the entire system as a whole. Refusing to create a new block for your platoon will result in a lower trust value and possibly access to limited maneuvers within a platoon. With this shift in the consensus algorithm away from POW, the proposed scheme isn't vulnerable to these attacks.

4.2.2. Applications of Blockchains. Many different applications have already been proposed for blockchains. Some of these blockchain applications are for VANETs. However, all of the applications suffer from the problems previously stated in Section 3.

In (Sanseverino *et al.*, 2017), the authors present a distributed solution for recording energy transactions between producers and consumers in a transactive energy management system. In this application, users can sell energy between themselves and to back to large scale producers, i.e. a utility company. The author was able to successfully detail how these transactions could be recorded on a blockchain. However, the authors admit that their model fails to capture the entire physical aspect of the energy transactions and the subsequent effects of any energy transactions on the entire system. Their particular solution suffers from cyber-only transactions that fail to represent the physical world in a verifiable way.

Multiple different papers have proposed applications that involve VANETs (Dorri *et al.*, 2017; Leiding *et al.*, 2016; Lu *et al.*, 2018; Roy and Madria, 2020; Singh and Kim, 2018; Xie *et al.*, 2019; Yuan and Wang, 2016). All of these papers propose solutions to secure VANETs with blockchains and management vehicles across the entire system. Additionally, they all involve a global blockchain for the entire network. These authors also rely heavily on costly infrastructure components such as RSUs and infrastructure entities such as called Overlay Block Managers. These approaches all fall victim to the real-time transaction requirements, a disconnected network, and cyber-only transactions that were

discussed in Section 3. These papers assume that the RSUs and other trusted components will be able to verify transactions. This is an incorrect assumption since they give no way for these entities to know the ground truth of the system. In a VANET, a transaction about road conditions at one location cannot necessarily be tied to vehicles next transaction in terms of verifiability. This is due to the randomness at which accidents, traffic, etc can occur on the roadways. So, taking that verifiability approach for VANETs cannot work.

The solution proposed in this dissertation doesn't suffer from any of these issues. Instead, it was specifically designed to address them. In the proposed system, vehicles use their sensors to verify the ground truth. There is no reliance on any infrastructure components with platoons managing a branching blockchain network themselves. This concept also addresses the need to maintain a blockchain across an entire network.

Other papers propose solutions that apply blockchains to different parts of VANETs outside of the normal driving applications (Gao *et al.*, 2018; Sharma *et al.*, 2017). The authors of (Sharma *et al.*, 2017) present a resource discovery and sharing framework for VANETs and smart cities. On the other hand, (Gao *et al.*, 2018) presents a privacy-preserving vehicle-to-grid payment mechanism for electric vehicle charging. Both of these papers address a less time-sensitive application. Additionally, for these applications, it makes sense that a city-wide blockchain is maintained. However, they don't address security in VANETs.

In (Roy and Madria, 2020), Roy et. al. proposed a scheme to secure VANETs for traffic management applications. They applied blockchains to allow vehicles to share information about traffic events to reduce congestion and improve roadway efficiency. The scheme is designed to handle a majority of malicious nodes. However, they rely heavily on RSUs similarly to other security approaches for VANETs.

The authors of (Singh and Kim, 2017) presented a reward-based intelligent vehicle communication that allows for authorized organizations to access the reputations of vehicles based on their past behavior. They present the idea of proof of driving which is a method for

verifying and validating the vehicles involved in VANET communication. However, they fail to explain what proof of driving is and why it will be an effective method for reaching consensus. However, all vehicles need to access a vehicle's driving history rather than just authorized organizations. If vehicles are going to interact and cooperate for a dangerous maneuver then they need to be able to have some way to prove that they are trustworthy. The proposed system takes a similar approach but uses their past behaviors as a token to join any need platoons and take full advantage of the different driving maneuvers in VANETs, i.e. tailgating.

4.2.3. Alternative Consensus Mechanism. BTC's property of "one CPU, one vote" restricts the number of votes a user has to the amount of money they can spend to buy computing power. However, they don't necessarily restrict a Sybil attack completely. A Sybil attack is when a user creates multiple fake identities. For example, a large corporation, government, or wealthy individual can leverage their money and resources to mount a Sybil attack and take over BTC by creating numerous nodes that all belong to the same entity. This could lead to a complete collapse in the cryptocurrency or any that have the same foundation as BTC. Multiple different mechanisms have been proposed to try to counter the issues faced by BTC's PoW consensus mechanism (Borge *et al.*, 2017; Wang *et al.*, 2018). However, each of these new solutions has its issues.

In (Borge *et al.*, 2017) they present Proof-of-Personhood as a way to overcome the "one CPU, one vote" concept. Instead, they use a "one person, one vote" system. Proof-of-Personhood is a mechanism that binds physical entities to virtual identities in a way that enables accountability while preserving anonymity. Their idea is to link virtual and physical identities in a real-world gathering while preserving user anonymity. However, a key issue for this protocol is that it is suggested for a permissionless cryptocurrency, i.e. anyone can join. In their system, the certification event and CA are both subject to attacks. Additionally, if there is a CA, even if it is a group of the users, then this makes it a permissioned blockchain in a sense since only certain users are allowed to mine, and

thus control the entire blockchain. This method may be a good idea for purpose large-scale government-backed cryptocurrencies in the future but as of now, it seems like a poor method for permissionless cryptocurrencies.

Another consensus protocol called Proof-of-Burn (PoB) was presented in (Wang *et al.*, 2018). In PoB, the right to mine a block and receive the mining reward is determined by the node who is willing to burn the more coin. Burning coins means that they are sent to an inaccessible address. Thus, once they are sent to that address they are no longer able to be used in the entire system. However, the main critique of this approach is that it benefits those who already have a large number of coins and can continue burning their coins to get permission to get create new blocks and determine the transactions on the blockchain.

A similar approach to PoB is PoS (Wang *et al.*, 2018). This approach works by deciding the miner for the next block based on the percentage of the entire net worth a single entity owns. This means that if one entity owns 50% of the stake in the network then they will get to mine 50% of the blocks. This approach suffers from the same criticisms of PoB in that it benefits those users who are already wealthy. Both PoB and PoS are a step away from the "one CPU, one vote" envisioned by Bitcoin's creators.

The proof-of-event (PoE) consensus algorithms proposed to validate traffic events in a VANET (Yang *et al.*, 2019). This algorithm is run by RSUs and collects state information from passing vehicles. Once a threshold value is hit, the event is claim as true and broadcast to the rest of the VANET. All valid traffic events are published to a blockchain with the proof used to validate them.

The algorithm proposed in this dissertation overcomes these issues faced by other consensus mechanisms for permissionless blockchains. In the proposed solution, vehicles are subdivided into a smaller group that needs to reach consensus - platoons. Since a platoon is limited in size, it can accommodate normal byzantine fault-tolerant consensus mechanisms without a huge impact on the time requirement (Castro *et al.*, 1999). Thus, it takes a "one vehicle, one vote" approach to consensus.

5. FORMAL VERIFICATION OF CYBER-PHYSICAL BLOCKCHAIN TRANSACTIONS IN VANETS

The proposed architecture differs from BTC's blockchain architecture in an attempt to solve the problems mentioned in the previous section. To apply blockchains to a real-time VANET in a disconnected network with a large degree of node mobility, some of the basic components of BTC's blockchain are altered. This architecture proposes a fundamental change to transactions and blockchain management as well as the addition of a CA. These proposed changes address the issues of verifiability and scalability when expanding the applications of blockchains.

5.1. TRANSACTIONS

Transactions in the proposed architecture are drastically different than BTC and other blockchains. In this system, transactions no longer transfer coins between users. They contain evaluations of the physical actions of other users, i.e. did the other user follow the guidelines of the system and any commands that were issued to them? The transactions are a cyber representation of the physical view of vehicles. This change drastically alters the verification process of the transactions. There is no way to verify transactions in the long term outside of verifying those who created the transaction. Instead, transactions must be verified by other users who witness the same physical actions, i.e. vehicles in a platoon in a VANET.

For example, during a lane-changing maneuver in a VANET, the vehicle making space for the merging vehicle needs to verify its size, speed, and any changes ahead in the roadway. Additionally, the vehicle needs to check that a vehicle is next to them trying to merge and that they are not under a bogus information attack. Even though some of this information can be collected from other sources, these vehicles need to verify it for

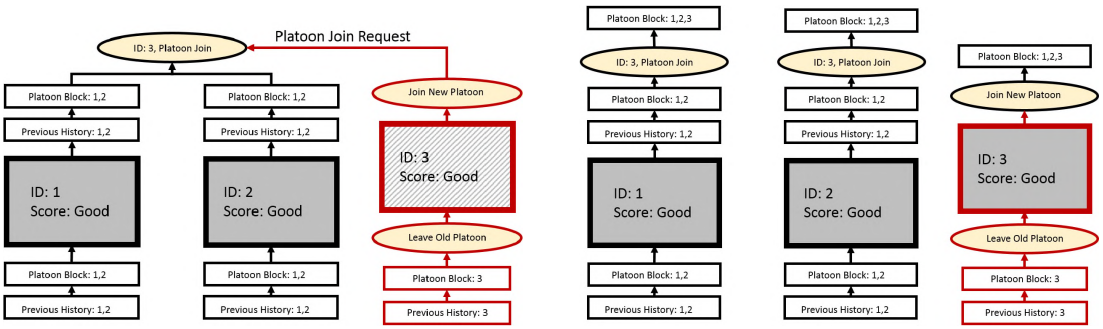
themselves. This means that their sensors are important to be able to sense what is going on around them and allows these vehicles to verify other's maneuvers. It is important to note that without an initial verification of the sensors by the CA, a vehicle could be running around blind which might risk the entire system. Thus, although the CA may not be central to day-to-day communication, it is required to extend some initial trust to vehicles before they have a history for other vehicles to rely on.

In the proposed system, the platoon leader of individual platoons is charged with issuing driving commands to the platoon. An example is when a platoon leader issues a slow down command to their platoon. All of the vehicles within the platoon then create evaluation transactions that denote what their sensors saw other vehicles do i.e did the other vehicles follow the command or is there something wrong with the vehicle? The amalgamation of these evaluations serves as a cyber verification of the physical system. Additionally, since vehicles are required to be within a platoon to create new blocks for their blockchain, a majority of a platoon must agree to cheat the system. But, if a vehicle does this often then the CA should have the ability to notice this and revoke the vehicle.

In terms of trust dynamics, a good evaluation can be seen as depreciating in the long term since future users cannot verify the transaction in the longer term. However, the proposed system requires users to create new blocks to be able to continue to participate. This solves the issue of the ability to forget which is mentioned in the next section.

Another interesting change is that transactions do not need previous transactions to validate future transactions. This leads us to the possibility of truncating our transactions and blockchain which Bitcoin cannot do since it needs to keep the records of any addresses that still have coins no matter where they are within a blockchain. Additionally, this change allows us to alter the storage of data across the entire network which results in our solution to the scalability issue mentioned in the next section.

The purposed cyber-physical transactions are a good first step to extending the property of data verification to other applications.



(a) The blockchains of vehicles 1, 2, and 3 before a platoon join is complete.

(b) The blockchains of vehicles 1, 2, and 3 after a platoon join is complete.

Figure 5.1. The blockchains of platoon vehicles before and after a join maneuver.

5.2. BLOCKCHAIN MANAGEMENT

In previous blockchain architectures, every user is required to possess the entire blockchain due to the need to validate all future transactions based on previous transactions. However, now that we have removed that need, the proposed solution opens itself up to a realm of different possibilities. In a VANET, users only need to know what the users in their platoon have done in the past. A vehicle in one section of the network doesn't care what other vehicles are doing in another section since their physical actions do not impact that vehicle. This allows us to create the cyber-physically partitioned blockchain. In the proposed architecture, every user only logs the transactions and blocks that it can verify. For example, a vehicle will only maintain the transactions and blocks that a platoon it is apart of has generated. Blocks in this application signify an agreed-upon state for the platoon. This log of transactions serves as a certificate of that vehicle's integrity when it joins a new platoon.

An example of this difference can be seen in Figure 5.1a and Figure 5.1b. These two images show a sample representation of the blockchains of different vehicles. Figure 5.1a shows the blockchains of vehicle 1,2,3 before vehicle 3 joins a platoon that consists of vehicles 1 and 2. Figure 5.1b shows all 3 blockchains after the platoon join has been

completed. Vehicles from different platoons will contain different information in their blockchain. However, once they are in the same platoon their blockchains will contain the same information during this time. When a vehicle decides to leave a platoon, a split will occur in the overall view of the network and the vehicles will once again maintain different blockchains.

The authors argue that this merging and splitting of blockchains makes sense due to the data verifiability property of the proposed scheme. Like Bitcoin, the users in this scheme only hold onto data they can verify. Since they are not physically present in other platoons, they are unable to verify the data they generate. This leads to the short-term verification nature of the transactions and the need for the CA to provide an initial certificate of trust. Additionally, vehicles are required to generate new blocks to continue to operate in a system. One issue with this approach is a user's ability to forget. This means that the other vehicles in a platoon cannot force that vehicle to store the block in memory. This issue is address by requiring users to generate new blocks based on the distance traveled. With this stipulation, a bad actor will suffer the same consequences whether they choose not to store a block or behave badly.

This proposed scheme uses this new blockchain management architecture to address the scalability issue when trying to apply blockchains to VANETs. It steps away from requiring a single global ledger and leaves us with the task of requiring separate platoons to meet real-time requirements when agreeing on the state of the platoon.

5.3. CERTIFICATE AUTHORITY

Another proposed change to Bitcoin's architecture is the addition of a CA. This drastically changes the semantics of the system as it transitions from a permissionless blockchain to a permissioned blockchain (Xu *et al.*, 2017). The CA performs two major actions: registering users and auditing the system.

As previously mentioned, in Bitcoin there are no negatives to not registering and verifying users since the system is self-governing and there is no need to link it to the physical world. Additionally, Bitcoin is purposefully designed to limit the impact of any single user by relying on the PoW mechanism and taking the "one CPU, one vote" approach. However, in CPSs, there is a need to link the cyber to the physical and verify that things are what they say they are and meet any laws of the physical system. This takes the form of a registered key pair that is used to sign all evaluation transactions generated by and of a particular vehicle. Although this generates a privacy concern, other vehicles need to be able to verify the identity of a vehicle when they present their blockchains for verification. If a vehicle generates a new key pair for every platoon, this task becomes impossible.

Another important aspect of registering is that the CA is initially certifying the integrity of the vehicles that register. Thus, other vehicles can trust that the certified vehicle will behave appropriately and can participate in the system. If vehicles were not registered then every vehicle would be required to initially assume that every other vehicle is hostile. This would result in a decreased initial efficiency of the system.

Secondly, the auditing function is a very important aspect of the proposed architecture since every user ends up with a different blockchain. Thus, an entity with the ability to see the bigger picture is important when protecting the system against different attacks. Even though the CA is not actively present in communications between vehicles, we can leverage its authority to identify anomalies and inappropriate actions. For example, it is assumed the CA can perform anomaly detection across all vehicles' blockchains to determine if a group of bad actors comes together to falsify a good history.

5.4. SECURITY THREAT MODEL

In the proposed architecture, there is a CA and a network of vehicles. In our security analysis, we assume that the CA is trusted while the network of vehicles is untrusted. This means that the CA will follow all protocols and not attempt to leak any information or

cheat. On the other hand, this dissertation assumes that the vehicle has the capacity to try to cheat the system or gain additional information. They have the ability to attempt to disrupt the system with their physical actions or false messages. This can take the form of an impersonation attack, message tampering, and even a bogus information attack. However, it is assumed that there is a bounded number of bad actors and that not all vehicles in the system are bad actors. In Section 5.6 some attacks are explored on the proposed scheme.

Now that the proposed architecture changes have been presented, a scheme for preserving security VANETs in the absence of RSUs is presented.

5.5. PROPOSED SCHEME

The proposed scheme applies the previously mentioned changes to overcome verifiability and scalability issues in VANETs. Both Sections 4.1 and 4.2 were published in (Wagner and McMillin, 2018). The goal of this scheme is for all vehicles to verify the integrity of one another without the need to constantly communicate with any sort of infrastructure. Vehicles within the system are split into two separate groups:

- Normal vehicles are vehicles with a trust value above a specified threshold. These vehicles operate normally within the system and are allowed to take part in risky maneuvers such as tailgating or bumper-to-bumper merging.

- Restricted vehicles are vehicles with a trust value below a specified threshold. These vehicles are allowed to be active within the system but have reduced capabilities. For example, they are not allowed to evaluate other vehicles or send commands to the platoon. However, they are allowed to receive and act on a specific subset of the commands and be evaluated by other vehicles. For example, they can perform a slow down maneuver but are not allowed to participate in a tailgate maneuver. This allows the vehicle a path to rebuild trust and reverify their integrity within the system without needing to recertify with the CA.

In the proposed scheme is it difficult for vehicles to determine whether a vehicle has been offline or has purposefully deleted parts of their blockchain. Thus, vehicles should be required to constantly update their blockchain to continue to prove their integrity. So, in the proposed scheme vehicles that do not have a new block in a certain distance traveled are also put on restricted status. An edge case could occur where it would be difficult for a platoon to tell the difference between a vehicle that has simply sat offline at a location for multiple days from a vehicle that is purposefully forgetting blocks. To overcome this issue, a secure Odometer or GPS is assumed to be used that tells the platoon the distance traveled since its last block was generated. The approach of using a trusted component was suggested in (Guette and Bryce, 2008). However, this work only uses a trusted component to reinforce our approach and solve a few issues rather than trusting it completely for our entire scheme like in previous work.

Additionally, since anything can happen on the road, it is difficult for a platoon to tell whether a vehicle is an adversary that is trying to penetrate and attack the system or just a vehicle that is having some technical difficulties. For example, if a vehicle gets dirt on some of their sensors and begins giving bad evaluations of other vehicles in the platoon even though other vehicles are giving good evaluations, the platoon may not be able to tell what is wrong with that vehicle. Thus, by allowing them to partially participate in the system without revoking them, it allows them to fix the problem without needing to communicate with the CA. This is an integrity solution rather than a security solution. If this were a security solution, any vehicle that makes a mistake should be revoked from interacting with other vehicles and treated as a malicious user.

In the proposed scheme there are three different types of transactions: evaluation transactions, command transactions, and join transactions. A command transaction is used to send commands to the platoon as well as store these commands. This transaction includes a signature, timestamp, and command. A sample command transaction can be seen in Figure

5.2. This transaction type is used when determining whether a platoon leader is a bad actor. This command transaction is referenced in the evaluation transaction and can be used to note that a platoon leader issued an invalid or inappropriate command.

An evaluation transaction is used to send evaluations of other's physical actions based on a command transaction. It includes the evaluator's signature, the trust score, a timestamp, the evaluated vehicle ID, and the evaluated command as seen in Figure 5.3. The evaluated command is just the command that the evaluator is basing its score on. Including this information can help with auditing by allowing the CA to denote specific issues with vehicles if they fail to follow a specific command. Lastly, a join transaction is used to transcribe a platoon join event. The join transaction includes the joining vehicle's certificate, a hash of their blockchain, a time stamp, and the joining platoon's leader. An example of a join transaction can be seen in Figure 5.4. It serves as a cyber representation that the physical action of joining the platoon was successful and met the relevant requirements such as being within a certain distance of the platoon. The join transactions allow a continued hash chain when a vehicle joins a new platoon since their blockchain's hash is now in their next block in the form of a transaction instead of in the header, unlike Bitcoin. This allows the vehicles to uphold the verifiability of the order of events like Bitcoin's blockchain. Note that including the hash of the blockchain allows the CA to determine if the vehicle is deleting part of their history when they perform a system audit.


```
{
'sender': 'http://localhost:5002',
'command': 'Platoon Slow Down',
'signature':
b'B\\a\xcb\x87m\x9f;\xc0\xfa\x16\n\xc2\xee\xb7\xfe[F\xac\xe6\xff\x82\x81\x8d\x8d\xcd~\x
17l\xad\xbd\xafr\xefU)#x\xe2S|YN\x9c\tS_\xb8\xa6Ec\x8d\x06Bl\x9c8\xbd\xf4\x1c\x12!L\x
80e\xbb;~c\xef\xd9\xe7a\x03l$O\x0e\xbeE\x91\x01\x02\xc7H\x85$?:n\x9b;QU,\x81\x04\xe
f\x03u\xe4e\x1a\xd7\x0c\x8b\xe8\xc5\xe5\xab@\x8c\x854T\xf7\x9fc\xeaRg\x9a`\xbe\xe9\x
96O'
}
```


Figure 5.2. Command Transaction Example

```
{
'sender': 'http://localhost:5002',
'recipient': 'http://localhost:5001',
'amount': 1,
'time_stamp': 1528725844.8794222,
'evaluated_command': 'Platoon Slow Down',
'signature':
b"y\xe8\x1a\xc3\xbc\xaa\x8f\xcdr\xe1D\xd3\x15J\xd2\xf7\xb9(\xda\xcb&\x91\xa9\xd7\xfd\
xd0\xdd\xd4\x7f=a\x16\x0fc\xcd\xd8\xee\xfa\xa1\xaa\xf3\xd4\x84\xfc\xc6vQ\x96\xe4P\xd8
\xe7U\x8d\x88/Z\xe4\x9c\xd8\x9b\xa5l\x8b\x89^$n?\x9e\xad\xc4\xd6\x97\xfd\x9f\xc21\xe
eW\x08\x1e\x92\xe7\xde5\xd0f\xe2\xf9Z\xe9\xdb\xfbb\x81\x1f\x12_\x81\xd2\xf6\x8eW'd,
\xd8\x10y\xac\xea\xeb_\xf3\x95]\x0c\xbbf\xce4\x06\xf0\xbdn\xe5v"
}
```


Figure 5.3. Evaluation Transaction Example


```
{
'index': 1,
'timestamp': 1528724787.226061,
'certification':
'cnZgcO723BBN7NraXKD9vmh8kpYCmL1USY58aGOVcZd4zfoA/tcX6w50zHEX0f+xiOyCLfYE/R
m3\nQ+F1wiTHc1Wm4ICFJdHgM2vXy00faGvOshIJAf80OArT1JO0k6njPs0IuLH8IRNi41TFfUi5Q
XUg\nxVRPWVuOjLwSHlendBY=\n',
'id':
'65537,10501110328004917724719021363868492439946312962430487016996374497129
9036681864338003882322412552850405437825868249341459955129020221455859936972
2969013799526974082868140623460581845182207029380791548678199534447129295540
4699240504842571135475242505348724882391556750310465160273061112603387222492
58536815717127'
}
```


Figure 5.4. Join Transaction Example

The scheme is split up into five different protocols: Car Registration, Block Creation, Join Platoon, Inner Platoon, and Leave Platoon. These protocols make up the basis for how the vehicles interact within a VANET. They are presented in the following subsections and verified in a later section. Additionally, how the CA performs System Auditing is briefly mentioned.

5.5.1. Car Registration. As previously discussed, registration is imperative to VANETs. The physical side of a vehicle needs to be verified before it can be allowed to participate in the system. Every vehicle must complete this protocol to receive a valid certificate and thus participate in the system. The key pair and genesis block, which is the first block in that vehicle's blockchain received, is then used in the rest of the system. Note that like all other transactions that go on a vehicle's blockchain, the genesis block that is created is a cyber certification of the physical readiness of the vehicle. A brief overview of the Car Registration Protocol can be seen in Figure 5.5. The Car Registration protocol works as follows:

1. The vehicle owner contacts the CA and meets them at a designated location. For example, this could be a Department of Motor Vehicles office or an automobile shop.

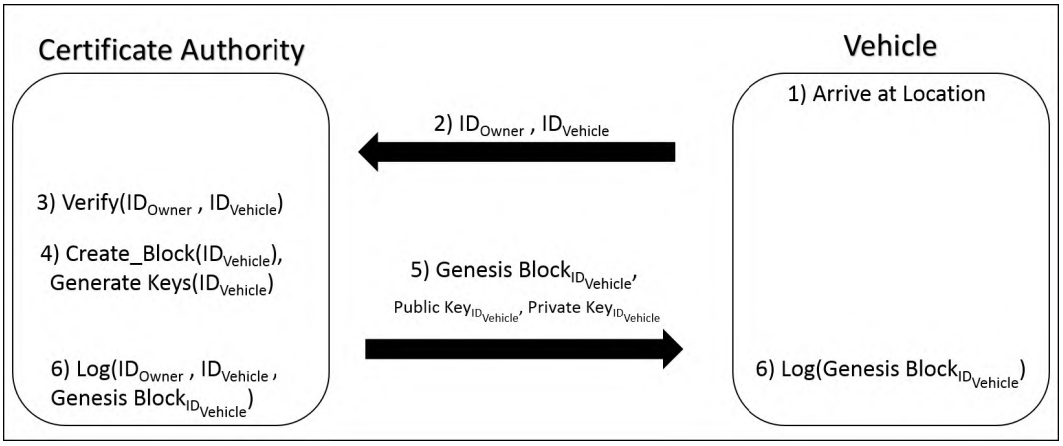


Figure 5.5. Car Registration Protocol

2. The vehicle owner gives the CA their private identification information as well as the car's identification information.

3. The CA verifies the vehicle's safety and roadworthiness as well as the owner's identity. Additionally, the CA can request taxes on the vehicle or for the owner to complete any other processes they wish.

4. If the information is valid and the vehicle meets all safety requirements then the CA creates a genesis block which includes a certification from the CA. Additionally, they create a public and private key pair so that the vehicle can sign their transactions.

5. The CA sends the key pair and the genesis block to the vehicle.

6. The vehicle and the CA both log the information received.

In the genesis block, the certificate is generated using the RSA digital signature scheme (Infomation Technology Laboratory and Technology, 2013). Here, the CA encrypts the public key of the vehicle, hashes it, then encrypts it with their private key. Note that any digital signature scheme can be used here. We assume that all vehicles have the CA's public key since all vehicles must get a genesis block from the CA to participate in the system. Thus, all vehicles should be able to verify the certificate of other vehicles. This scheme assumes that other vehicles can't recreate a CA certificate, thus creating their genesis block and certification.

An example of a genesis block can be seen in Figure 5.6. This block includes the certificate of the vehicle, the public key of the vehicle, the timestamp, and the block index.

5.5.2. Platoon Block Creation. This scheme relies on the integrity validation from the platoon join protocol to speed-up block creation. In particular, the scheme saves time and energy by making the platoon leader randomly select a platoon member to create a new block. No hash puzzle is solved since there is no benefit to creating a new block thus no need to compete for its creation. In this protocol, the platoon leader has to select a

{
'index': 1,
'timestamp': 1528724787.226061,
'certification':
'cnZgcO723BBN7NraXKD9vmh8kpYCmL1USY58aGOVcZd4zfoA/tcX6w50zHEX0f+xiOyCLfYE/R
m3\nQ+F1wiTHc1Wm4ICFJdHgM2vXy00faGvOshIJAf80OArT1JO0k6njPs0luLH8IRNi41TFfUi5Q
XUg\nxVRPWVuOjLwSHlendBY=\n',
'id':
'65537,105011103280049177247190213638684924399463129624304870169963744971299
036681864338003882322412552850405437825868249341459955129020221455859936972
296901379952697408286814062346058184518220702938079154867819953444712929540
469924050484257113547524250534872488239155675031046516027306111260338722249
258536815717127'
}

Figure 5.6. Genesis Block Example

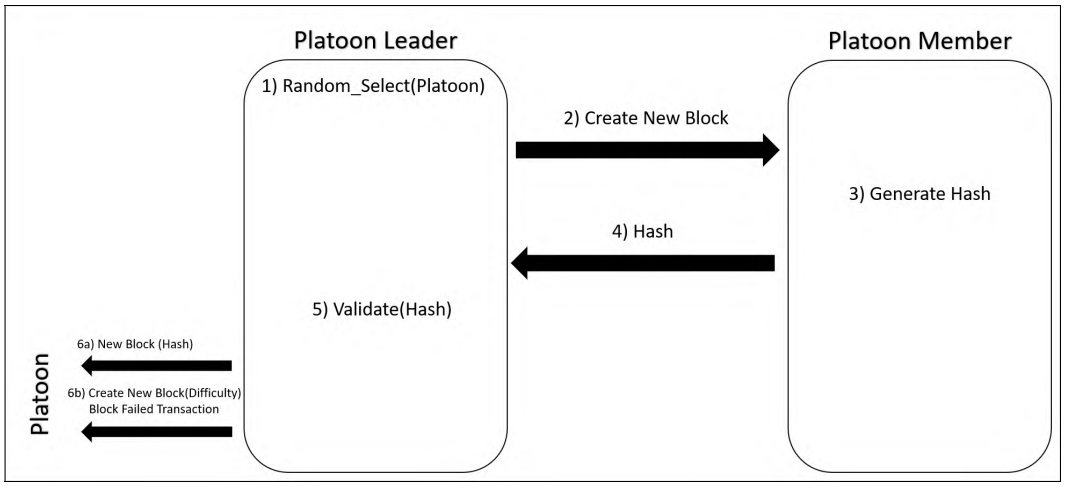


Figure 5.7. Platoon Block Creation Protocol

platoon member to create a block for the platoon. Once the block is created, it is sent back to the platoon leader, verified, then broadcast to the rest of the platoon. In the case of a failure to create a new block, a transaction is instead broadcast to the platoon and another member selected. The authors argue that this is acceptable due to the private nature of the blockchain. Vehicles are verified when they join a platoon and are constantly checked for their continued membership. Thus, their physical action of joining and participating in a platoon gives a vehicle permission to create new blocks. An overview of the Platoon Block Creation protocol can be seen in Figure 5.7.

1. The platoon leader randomly selects a platoon member that does not have a restricted status to create the next block of the platoon.

2. The platoon leader sends a **Create New Block** message to that member.

3. The chosen vehicle receives the **Create New Block** message and calculates the new hash.

4. When the vehicle generates the hash, it sends the nonce and the block to the platoon leader.

5. The platoon leader verifies the block.

6. (a) If no issues are found it sends a **New Block** message to the entire platoon with the Proof-of-Work attached.

   (b) If the block isn't correct, the platoon leader randomly selects a different vehicle and restarts the process. Additionally, a transaction noting the failure is created and sent to the platoon.

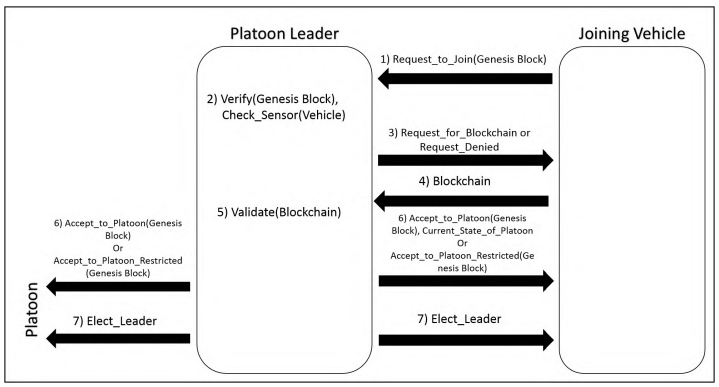


Figure 5.8. Platoon Join Protocol

This protocol is run as a state agreement protocol. Additionally, it serves as a token to prove that all of these physical maneuvers and the related transactions were physically verified by other users. The other vehicles are required to verify the block on receipt. Once a block is published, it can be used to show a vehicle's trustworthiness when they try to join another platoon.

5.5.3. Platoon Join. The Platoon Join protocol is run whenever a vehicle tries to join a platoon. The vehicle must be within the sensor range of the platoon to join the platoon so that the other vehicles can verify the physical process of joining a platoon. This is another example of using the physical properties of the application to verify the cyber transactions. The protocol works as follows:

1. A requesting vehicle sends a **Request to Join** message which includes their identification number and their generator block to the platoon leader.

2. The leader of the platoon receives the **Request to Join** message and verifies that the genesis block is valid. In particular, they check that the certificate in the generator block is valid and that the genesis block hasn't passed its expiration date. Additionally, the platoon leader verifies that the requesting vehicle is within its sensor range.

3. (a) If the genesis block is valid, the platoon leader sends a **Request for Blockchain** message.

(b) If the genesis block is not valid the platoon leader replies with a **Request Denied** message. Additionally, the platoon leader publishes a request denied transaction to the rest of the platoon that includes the identification information and the certificate of the requestor.

4. The requesting vehicle receives the **Request for Blockchain** message. They send their entire blockchain to the platoon leader.

5. The platoon leader then verifies the blockchain by validating the hash chain. Additionally, they calculate the vehicle's trust score by amalgamating the previous transactions that belong to the joining vehicle within their blockchain.

6. (a) If the vehicle's trust score is above the trust threshold then all vehicles are sent a **Accept to Platoon** message by the platoon leader. Additionally, the joining vehicle is sent the current state of the platoon. This includes any unpublished transactions as well as the last platoon block.

   (b) If the vehicle's trust score is below the trust threshold then the vehicle is allowed into the platoon on a restricted basis and all vehicles are sent a **Accept to Platoon Restricted** message by the platoon leader. They publish these messages to the platoon's chain.

7. If the vehicle's trust value is higher than the current platoon leader's trust value then they send a **Elect Leader** message to the entire platoon along with that vehicle's ID. The vehicle then hands over the platoon leadership to that vehicle.

5.5.4. Intra Platoon Communication. The Intra Platoon Communication protocol is run whenever a command is issued or at regular intervals to verify that the vehicles within the platoon are behaving correctly. An overview of this protocol can be seen in Figure 5.9. This protocol works as follows:

1. The platoon leader issues a command to the whole platoon.

2. Every vehicle verifies that the command is valid and won't put them in danger. Then the platoon attempts to follow the command.

3. Every vehicle in the platoon uses its external sensors to examine the other cars in the platoon to determine if they followed the command.

4. Every vehicle then creates a transaction evaluating the vehicles within their sensor range on their ability to follow the command.

5. The vehicle will send this to all of their platoon members.

6. After a certain time or after a certain event, such as a platoon join, the Platoon Block Creation protocol is run by the platoon leader.

7. After the protocol completes, for any vehicles that go below the trust threshold the platoon leader sends a **Put X on Restricted Status** message to the entire platoon as well as a transaction noting that the vehicle was given restricted status and stops receiving transactions from the vehicle.

8. If there is a higher trust score the Platoon leader every vehicle broadcasts an "Elect Leader" along with the vehicle's ID whose trust value is highest.

In this protocol, we give the individual vehicles a decent amount of autonomy when receiving commands from the platoon leader. This along with how each vehicle reacts to different commands and who is allowed to give commands to the platoon should be studied further in the future.

5.5.5. Platoon Leave. The Platoon Leave protocol is run every time a platoon member wants to leave the platoon. A diagram representing the order of actions taken in the Platoon Leave protocol is shown in Figure 5.10. The protocol works as follows:

1. A vehicle wants to leave the platoon. It sends a **Request to Leave** message to the platoon leader.

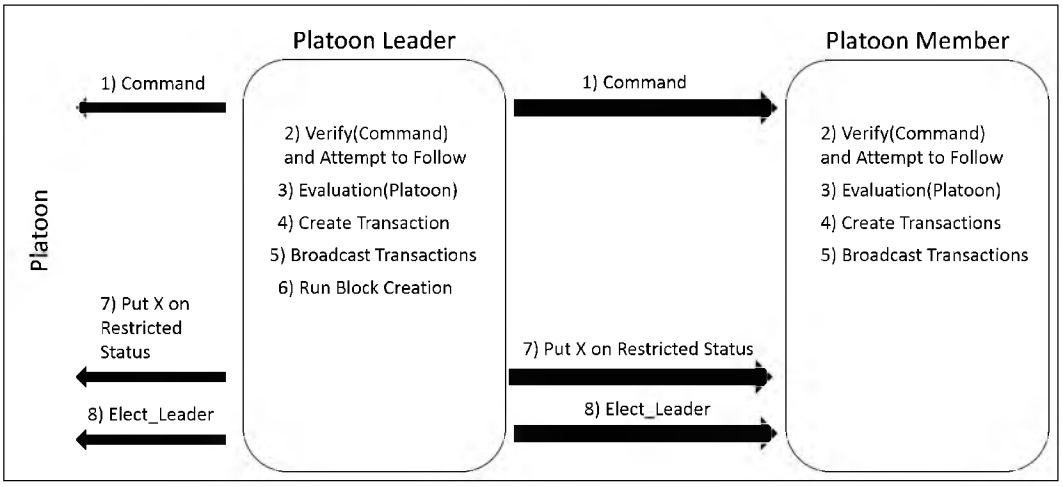


Figure 5.9. Intra Platoon Communication Protocol

2. The platoon leader sends an **Acknowledge** message to the leaving vehicle.

3. Once the vehicle has left the sensor range of the platoon, the platoon leader and other platoon members then send a transaction to the platoon noting that the vehicle is no longer within their sensor range.

4. The leaving vehicle deletes the unpublished transactions from its history.

An interesting note of this protocol is that it is resilient in the face of dropped messages. If either the **Request to Leave** message or **Acknowledge** message gets dropped then, due to the physical nature of the system, the platoon will know that the platoon member has left the platoon. However, the platoon will note that something didn't happen correctly when they make evaluation transactions. This is a major benefit of physically verifying the physical actions within the platoon.

5.5.6. Trust Scores. Whenever a maneuver is performed by the platoon, an evaluation is created that signifies the ability for a vehicle to properly obey commands. Evaluations can grade either cyber or physical portion of a vehicle. The cyber evaluations include the ability to properly communicate with the rest of the platoon, the ability to create a correct new block, and the overall ability to follow commands of the platoon. The physical evalua-

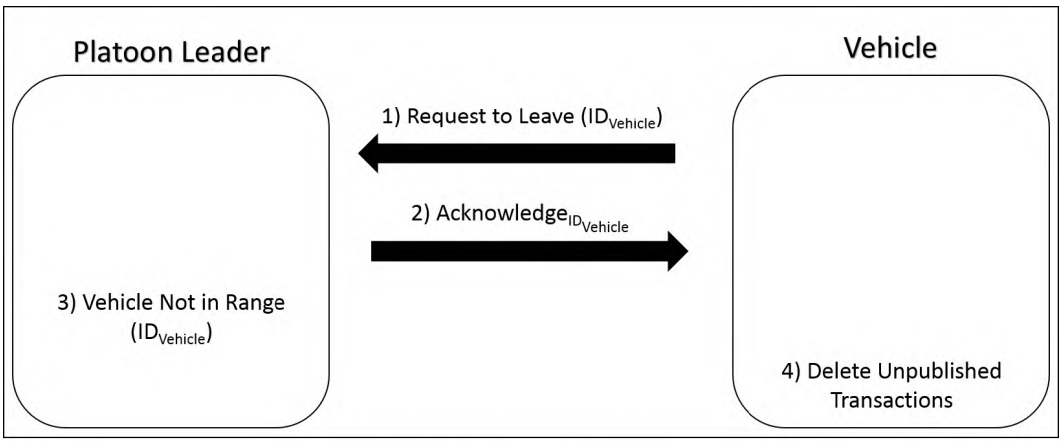


Figure 5.10. Platoon Leave Protocol

tion grade the physical actions of a vehicle. This includes their acceleration, deacceleration, speed, turning ability, and ability to perform general maneuvers such as a platoon join and leave. Scores will be a value of one or zero. One denoted that the vehicle behaved according to the rules of the system whereas a zero denotes a bad action. The overall trust score of a vehicle is calculated using the equation shown below. Here N is the number of scores in their blockchain, S_k is the evaluation of the vehicle, B_C is the current block, and B_S is the block of evaluation S. Thus, the overall trust score is:

$$\frac{\sum_{k=1}^{N} \frac{S_k}{(B_C - B_S + 1)^2}}{N} \tag{5.1}$$

The trust score of a vehicle is a sum of their overall history performing in the system. The value of evaluations degrades exponentially over time. Thus, the system will value the most recent evaluations more than older evaluations. The final trust score is the average of all of these sums. This trust score calculation was modeled off the work done in (Duan and Chow, 2018). To join a platoon, a vehicle needs to be above a threshold of 0.98. This threshold allows vehicles to recover their trust score and continue to participate in the system in the face of a general fault. For example, if a vehicle has a sensor that has a fault but recovers, then the vehicle can continue to participate without being put on restricted status

5.5.7. System Audit. Over time the CA needs to audit the system to ensure that vehicles are not deleting parts of their history, creating false history within a platoon of other malicious vehicles, etc. To ensure that these actions are not happening they need to have to ability to audit the system. To audit the system, the CA just needs to collect all of the vehicle's blockchains up until the current time that they need to verify. We assume that they can do this on command, in the case of an emergency, or whenever a vehicle needs to get recertified, in case of periodic inspections of the physical vehicles. Then, the CA can piece individual blockchains together to generate the view of the entire network to verify that everything is working correctly. Since multiple vehicles will have the same blocks as the other vehicles within a platoon, it should be relatively easy for a CA to match all of the blockchains together to be able to track vehicles and their behaviors throughout the system. Once history from many vehicles is pieced together, the CA just needs to find anomalies or other issues.

5.6. SECURITY ANALYSIS

This dissertation proposes a method for vehicles to verify one another's integrity without the constant assistance of a third party. Many attacks have already been proposed in VANETs. These include bogus information injections, denial of service, impersonation, Sybil attacks, message suspension, message tampering, location faking, and more (Zeadally *et al.*, 2012) (Qu *et al.*, 2015). However, due to the proposed scheme's focus on integrity verification, only bogus information injections are covered.

A bogus information injection occurs when a vehicle sends other vehicles incorrect information. This can either be something as small as an incorrect trust score or as large as a false command, i.e. an emergency brake alert. This type of attack can be done maliciously or accidentally. In the proposed scheme, such an attack is mitigated via the sensing of off-chain actions by vehicles within a platoon.

In the Platoon Join protocol, a vehicle could attempt a bogus information injection by trying to join a platoon without being near the platoon. The attacking vehicle would send the **Request to Join** message to the platoon initiating the protocol. However, Step 2 in the protocol requires the attacking vehicle to be in the sensing range of the platoon for the protocol to continue. Thus, the Join Platoon request would be rejected by the platoon since the joining vehicle is within the sensing range.

A similar scenario occurs in the Leave Platoon protocol if an attacking vehicle tells the platoon they are leaving the platoon and then doesn't leave. In this protocol, Step 3 only occurs once the vehicle has left the sensor range of the platoon. This means that the vehicle cannot leave the platoon unless they physically leave the platoon's sensor range. The attack in this scenario would not succeed.

In the Intra Platoon Communication protocol, there are several places for a bogus information injection by an attacking vehicle. In this protocol, the platoon leader could issue an invalid command in Step 1, a vehicle can issue an incorrect trust evaluation in Step 4, and the platoon leader could put a good vehicle on restricted status in Step 7. Due to Step 2, the vehicles verify any commands from the platoon leader. This means that the vehicle makes sure the command will not harm the vehicle. For example, if the platoon leader tells the platoon to turn left off a cliff, the vehicle will use their sensing equipment to note that the action would result in a dangerous outcome and not complete the command, mitigating the attack. In Step 4, if a single vehicle issues an incorrect trust evaluation, the trust evaluation will be out-weighed by the rest of the platoon. This results in the attack having little to no effect on the evaluated vehicle's score. Lastly, if the platoon leader attempts to put a good vehicle in restricted status the rest of the platoon will be able to verify the status via their blockchain. Thus, since all of their actions are already recorded on the blockchain, the rest of the platoon will know that the vehicle should not be on restricted status. From that point, they could elect a new platoon leader and note the acts of the old platoon leader via publishing negative evaluation transactions.

A denial of service (DOS) attack occurs when an action is taken that prevents the system from moving forward, essentially freezing it in its current state. In the proposed scheme, a denial of service attack can result from many different actions. For example, the platoon leader could refuse to initiate any protocols. If this were to occur, the platoon could simply elect a new platoon leader and evaluate the platoon leader accordingly. Another approach could be to remove the platoon leader entirely from the platoon and create a scheme where the vehicles vote on actions.

An impersonation attack occurs when one user pretends that they are another user. However, the proposed scheme is secure against this attack. It mentions the assumption that only the CA can create genesis blocks and a valid certification. This would prevent any attacker from being able to forge another vehicle's signature. Another approach could be for a vehicle owner to register multiple vehicles and transfer identity information among them. This can be considered a Sybil attack. However, due to the detection of the physical presence of a joining vehicle, an owner could only use one vehicle's credentials at a time. Additionally, the secure GPS would alert the platoon if the certification was outdated based on the distance the current vehicle has traveled. If an attacker tried to switch their credentials between different vehicles, the platoon would be alerted by the secure GPS. Thus, the attack would fail.

Message suspension is when a vehicle refuses to forward a message or vehicles refuse to communicate with one another. In the proposed scheme, the biggest concern is when a vehicle join or vehicle leave message is sent to the platoon that it is propagated to the rest of the platoon. In the case of the vehicle join, eventually, the vehicle should merge with the platoon which would result in multiple vehicles receiving the message, mitigating the attack. Additionally, basic platoon restrictions, such as all vehicles needing to be within one-hop broadcast range, could help mitigate this problem. On the other hand, the vehicle leave protocol mitigates the message suspension attack. As long as the vehicles sense that a vehicle has left the platoon the protocol can complete.

Message tampering is when a vehicle changes another vehicle's message when it forwards it to another part of the platoon. Due to the way transactions are created, message tampering is impossible. Thanks to the digital signatures implemented by Bitcoin's blockchain that is also implemented here, this attack cannot occur without a vehicle first stealing another vehicle's private information. However, even though the vehicles are untrusted we can safely assume they will not give out their private information. Thus, this type of attack is mitigated by the proposed system.

In the proposed protocol, the sensing capabilities of the platoon and each vehicle allow the entire scheme to be secure again bogus information attacks. These bogus information attacks are mitigated by the protocol. In the next section, the inherent traits and security properties of the protocols are turned into invariants of the system. These invariants are then formally verified. Thus, the security of our system against these ad hoc bogus information attacks are formally verified.

5.7. FORMAL MODEL VERIFICATION [1]

To show the correctness of the proposed communication model, we performed system verification with the SPIN model checker. This section is in the submitted journal version of this work. This tool works by thoroughly checking the states that are generated from a distributed system design. (Holzmann, 2004) The user first constructs a verification model that has all the required system properties. In general, this will be abstracted from the true mathematical/complex operations that occur in the actual design. Next, the user generates a list of neverclaims which are representations of invariants for the model. Invariants are properties that must hold for the entire system life-cycle and are represented

[1]This section was part of a journal paper submitted to the International Journal of Critical Infrastructure Protection (IJCIP).

using LTL. SPIN then system states that are generated from the verification model against the neverclaims and returns an error if there exists a state where the neverclaims became false.

One issue with model checking is the existence of the state explosion problem. This is the exponential increase in the number of states due to an increase in the number of global state variables and possible branches in each process in a concurrent system. (Clarke *et al.*, 2011) To overcome this obstacle, three things were done. First, the presented communication protocols were split into three protocols instead of one entire procedure; the join protocol, the leave protocol, and the intra-communication protocol. The Promela code created for all three protocols can be found in Appendix A. The intra-communication protocol consists of the Intra Platoon Communication protocol and the Block Creation protocol. Next, the complexity of the verification model was reduced by removing any complex computations and instead abstracting these ideas to state variables. For example, the model does not contain a blockchain or and blocks. Instead, these ideas were abstracted to state variables representing the fitness of individual vehicles. Lastly, the Swarm mode in SPIN was used. (Holzmann *et al.*, 2010) The Swarm mode of SPIN uses the ideas of parallelism and search diversity to try to solve the state explosion problem. In particular, it performs a range of different search methods to run many small verification jobs in parallel in a partial state-space search.(Holzmann *et al.*, 2010) They argue that by checking many different areas of the search space, it is highly likely that a majority of the errors can be found without a full state-space search. Now that we have introduced the methodology of the formal model verification, we present our work.

5.7.1. Invariants. The invariants we generated reflect what should happen within the communication protocols and the physical actions within them. However, it does not cover what vehicles should look for in terms of correct or incorrect actions. For example, the invariants mention that a vehicle should be present to join a platoon. But, it fails to mention what a vehicle should look for when evaluating other vehicles for a particular

Table 5.1. Table of Symbols

Symbol	Definition
$\Box$	the LTL always operator
$\Diamond$	the LTL eventually operator
$\neg$, $\wedge$, $\vee$	the boolean negation, and, or logical operator
$\bigcup$	the temporal strong until operator
$\rightarrow$	the boolean logical implication operatory
$\leftrightarrow$	the boolean logical equivalence operator
B_m	A bad maneuver was performed by a vehicle.
BR_{Join}/BR_{Leave}	A vehicle broadcasts a requst to join or leave
IN_p	A vehicle is in a platoon.
C_p/C_n	A vehicle has either a recent positive or negative recertification block.
Phy	A vehicle is physically present within the platoon.
BC	A block creation event is performed by the platoon.
$P_j/P_{jr}/P_l$	A vehicle joins, joining on restricted status, or leaves the platoon.
GPS	A vehicle's secure GPS reports a valid answer.
$command$	A vehicle issues a command to the platoon.
$evaluate$	A vehicle creates an evaluation of a platoon member.

action. Instead, the readers should see (Kanteti, 2017) for work regarding the physical invariants within the platoon. A list of symbols used for our LTL invariants can be seen in Table 5.1. The 11 invariants that were created are now presented in LTL and written format. The Promela code for the invariants used by SPIN can be found in Appendix B.

Invariant 1: $\Box((B_m \wedge IN_p) \rightarrow \Diamond C_n)$

It is always true that if a vehicle makes a bad maneuver and is in a platoon then eventually the car will receive a negative certification block.

Invariant 2: $\Box((\neg B_m \wedge IN_p) \bigcup BC \rightarrow C_p$

It's always true that if a vehicle does not make a bad maneuver and is in a platoon until a block creation event then eventually it will receive a positive recertification block.

Invariant 3: $\Box((Phy \wedge C_p \wedge BR_{Join} \wedge GPS) \leftrightarrow P_j$

It is always true that if a vehicle is physically present, possesses a positive recertification, requests to join a platoon, and produces a correct GPS result then it will be allowed to join the platoon.

Invariant 4: $\Box((Phy \wedge C_n \wedge BR_{Join} \wedge GPS) \leftrightarrow P_{jr}$

It is always true that if a vehicle is physically present, has a negative recertification, requests to join a platoon, and produces a correct GPS result then it will be allowed to join the platoon.

Invariant 5: $\Box(P_j \rightarrow P_j \bigcup(C_n \vee BR_{Leave})$

It's always true that if a vehicle joins a platoon, then it will be apart of that platoon until it receives a negative recertification or broadcasts a request to leave the platoon.

Invariant 6: $\Box(P_{jr} \rightarrow P_{jr} \bigcup(C_p \vee BR_{Leave})$

It's always true that if a vehicle joins a platoon on restricted status, then it will be apart of that platoon on restricted status until it receives a positive recertification or broadcasts a request to leave the platoon.

Invariant 7: $\Box(P_j \vee P_{jr} \rightarrow \Diamond BC)$

It's always true that if a vehicle joins a platoon or joins a platoon on restricted status then eventually it should receive a new block.

Invariant 8: $\Box(BC \rightarrow C_n \vee C_p)$

It's always true that a blockchain creation event implies that a vehicle will receive a positive or negative recertification block.

Invariant 9: $\Box((P_j \vee P_{jr}) \rightarrow (\neg BR_{Join} \bigcup P_l))$

It's always true that if a vehicle joins a platoon or joins a platoon on restricted status then it cannot request to join another platoon until it leaves the current platoon.

Invariant 10: $\Box(C_n \rightarrow \neg(command \vee evaluate))$

It's always true that if a vehicle has a negative recertification then it cannot issue a command or evaluate a platoon member.

Table 5.2. SPIN Swarm Results for Invariants

38 Spin Verification Runs			
Invariant	1	3	11
Protocol	Intra Platoon	Join Platoon	Leave Platoon
State-Vector	808 - 816	956	792
Depth Reached	127 - 853	127 - 1302	127 - 557
States Stored	170 - 33,886,973	2,885 - 39,937,254	2,080 - 18,682,676
States Matched	61 - 48,881,998	3,564 - 53,011,589	1,676 - 9,791,978
Transitions	231 - 82,768,971	6,449 - 90,965,306	3,756 - 9,791,978
Atomic Steps	115 - 21,379,071	215,446 - 144,614,480	8,302 - 17,836,212

Invariant 11: $\Box(P_l \rightarrow (\neg Phy \wedge BR_{Leave}))$

It's always true that if a vehicle leaves the platoon then it is not physically present within the platoon and it broadcasts a request to leave the platoon.

These protocols attempt to describe some behavior that should be expected within a platoon. Invariant 1 and 2 describe the cases that should result in a positive or negative recertification block. Invariants 3 and 4 present the platoon join conditions for both restricted and normal status vehicles. Invariant 5 and 6 present the requirements to continue being a part of a platoon in a given status. Invariant 7 hints at the idea that whenever a vehicle joins a platoon they should be staying long enough to participate in a blockchain creation event. Invariant 8 simply means that if a vehicle participates in a blockchain creation event then they should receive a new negative or positive recertification. Invariant 9 pushes the idea that vehicles should not be able to join more than one platoon at a time, which is provable by their blockchain. Invariant 10 ensures vehicles on restricted status only do certain actions. Invariant 11 shows the conditions for leaving a platoon. All of these invariants were coded with SPIN and verified via the SWARM method.

5.7.2. SPIN Results. The communication protocol was split into three different sections: platoon join, intra-platoon communication, and platoon leave protocols. The Intra platoon communication included the steps for the block creation protocol. This division of protocols was done to deal with the state explosion problem. Additionally, each invariant was then applied to the relevant section of the protocol. Invariants 1, 2, 5, 6, 7, 8, 9, and 10 were tested over the intra-platoon communication protocol. Lastly, Invariant 11 was tested over the leave protocol. Table 5.2 shows the results of applying one invariant to each protocol. The values are approximately the same for each protocol regardless of which invariant is applied to it.

Swarm created 38 different state-space searches all with different parameters for each run. The state-vector size was between varied depending on which protocol was tested The state-vector size is the required memory to describe a single global system state. For each search, a report is created that includes the depth reached, states stored, states matched, transitions, atomic steps, and errors found. The errors refer to the invalidation of the model under the given neverclaim. For all of our invariants, no errors were generated. The depth reached is the longest depth-first search path. The states stored are the number of unique global system states. The number of states matches is the number of times a search returned to a previously visited state in the search tree. The transitions are the number of transitions that were explored in the search and can serve as a representation of the amount of work done to complete a given state-space search. Lastly, atomic steps are the number of steps that were carried out as part of an atomic sequence. In Spin, an atomic sequence is a series of steps that are performed as one step.

Swarm has twelve different types of searches it performs. These include four base searches: basic depth-first search (DFS), reversed process ordering DFS, reversed transition ordering DFS, reversed process ordering, and transition ordering DFS. Additionally, there is a randomized version and bounded context switching version of the four base searches. The number of different state space searches generated, which was 38, results from the

number of CPUs available and the amount of time available. For the model verification, a VM that has 4 CPUs and 16GB of RAM was used. Thus, 3 CPUs and 1 hour were allotted for each run. In Table 5.2, the runs with the smallest amount of depth and states happen to be the randomly generated runs.

The complete SPIN results can be found in Appendix B. Now that the communication protocol was proven correct under the given invariants, the results of a simulation are presented.

5.8. TIME ANALYSIS

In this section, the time cost of the protocols is compared to the cost if RSUs were included in the process. Additionally, it is verified that the protocols meet the real-time requirements for the physical maneuvers of the platoon. According to (Amoozadeh *et al.*, 2015), the cost of different platoon maneuvers can be seen in Table 5.3. Let M be the amount of data in a message divided by the bandwidth of the channel, S be the time for a vehicle to check it's sensors, B be the time to send an entire block of a vehicle's blockchain, K be the time to verify that a block is correct and calculate a trust score for a block, and H be the number of blocks in the entire history of a vehicle. Thus, the cost for Platoon Join protocol is:

$$2M + HB + HK + S \tag{5.2}$$

The cost of the Platoon Leave protocol is:

$$2M + S \tag{5.3}$$

No other protocols are analyzed since they are not time dependant on a platoon maneuver. It is assumed that the VANET is using a 5.9 GHz channel to communicate as stated in IEEE 802.11a and 11p (Hartenstein and Laberteaux, 2008). Additionally, it is

assumed that the transaction size is 500 Bytes as suggested by (Wiki, 2019) and the header size is 80 Bytes (Reference, 2019). These are both standard transaction sizes based on the current Bitcoin architecture. This dissertation assumes that a normal message is 1000 Bytes and that a platoon will only contain 10 vehicles. It is assumed that the time it takes to verify a single block is .0047s or 4.7ms.

To show that the proposed protocols satisfy real-time requirements, it must be shown that the Join Platoon protocol can be completed in the time it takes for the merge platoon maneuver while the Leave Platoon protocol can be completed in the time it takes for a leader to leave. The merge platoon maneuver was selected since a vehicle joining a platoon of size n vehicles can be seen as a platoon of size 1 merging with a platoon of size n. For the Platoon Leave protocol, the leader leave maneuver is chosen since it is the lowest acceptable time requirement.

$$16 > 2M + HB + HK + S \tag{5.4}$$

After factoring out H and moving everything to the left-hand size, the following is left:

$$\frac{16 - 2M - S}{B + K} > H \tag{5.5}$$

By plugging in all of the numbers from the aforementioned assumptions, the following is returned. Note that the channel speed is replaced with the theoretical channel speed rather than the real one. However, this is acceptable due to the definition of less than.

$$\frac{16 - \frac{2x10^3}{5.9x10^6} - S}{\frac{5080}{5.9x10^6} + 0.0047} > H \tag{5.6}$$

Table 5.3. Real-Time Cost of Platoon Maneuvers

Maneuver	Approximate Time (Seconds)
Merge	16
Split	4
Leader Leave	4
Last Follower Leave	4
Middle Follower Leave	8

By assuming some arbitrary value for S, i.e. 0.5, H must be less than 2788 blocks to meet the time criteria for the merge maneuver. Thus, as long as a vehicle gets recertified every 2788 blocks, it will be able to meet real-time requirements. To verify that the Platoon Leave protocol meets real-time requirements, the following is found assuming the same arbitrary value for S:

$$4 > 2M + S = \frac{2x10^3}{5.9x10^6} + 0.5 = .5004 \tag{5.7}$$

Thus, the protocol meets real-time requirements for both the Platoon Join and Platoon Leave protocols. Since those are the only two protocols constrained by platoon maneuvers, the entire proposed system meets real-time requirements.

5.9. SIMULATION RESULTS

A simulation of the cyber components of the proposed algorithm was created on a Dell Precision M4400, with 4.00 GB of RAM and an Intel Core 2 Extreme CPU Q9300, using Python.

The PoW puzzle was implemented to verify that a high difficulty factor cannot be used in the proposed solution. Figure 5.11 presents the time taken to solve the PoW puzzle based on a given difficulty. Based on the data, the time it takes to complete the puzzle is exponential to the difficulty factor. Just with a difficulty of six leading zeroes, it takes approximately 200 seconds to calculate the answer. Since actions within a VANET happen

in a matter of seconds if not faster, this speed is unacceptable. On a normal roadway, vehicles could join and leave a platoon in less than 200 seconds. This would result in no updated history for this time which could lead to cheating. However, our scheme does not encounter this issue since any hash value is accepted for a given block. Instead, it relies on the broadcast time for messages.

To create a certificate the RSA library was used which implements RSA Digital Signatures, RSA Key Generation, and RSA Encryption/Decryption. This code used 1024 bit RSA key size. Ten trials were run and the results were plotted in Figure 5.12. As shown in Figure 5.12, the time it takes for the certificate authority to generate a new key pair as well as create a certificate can vary from less than .5 seconds to almost 6 seconds. However, since they are being done when the vehicle is being certified by the CA and there is a physical vehicle check associated with the operation there are no time constraints. Thus, the times are acceptable.

The Join Platoon protocol was also simulated. Figure 5.13 shows the results of the implementation. With a block size of 100 transactions, the time it took to complete the protocol was tested. Based on the results, the time it takes to complete the protocol is linear to the number of blocks that need to be verified. This is since every transaction is checked to see if it is evaluating the joining vehicle and then the results are added together if it is. Based on this data, the protocol will take more time the longer a vehicle is participating. This results in the protocol eventually being to slow for a VANET. However, due to change in the way transactions or created and verified truncation or other time-saving operations could become viable.

The simulation results show the Platoon Join protocol has scalability issues in terms of the time it takes to join a platoon. The main issue is the time it takes to transmit an entire blockchain then verify it. PoW is shown to be an unacceptable consensus method

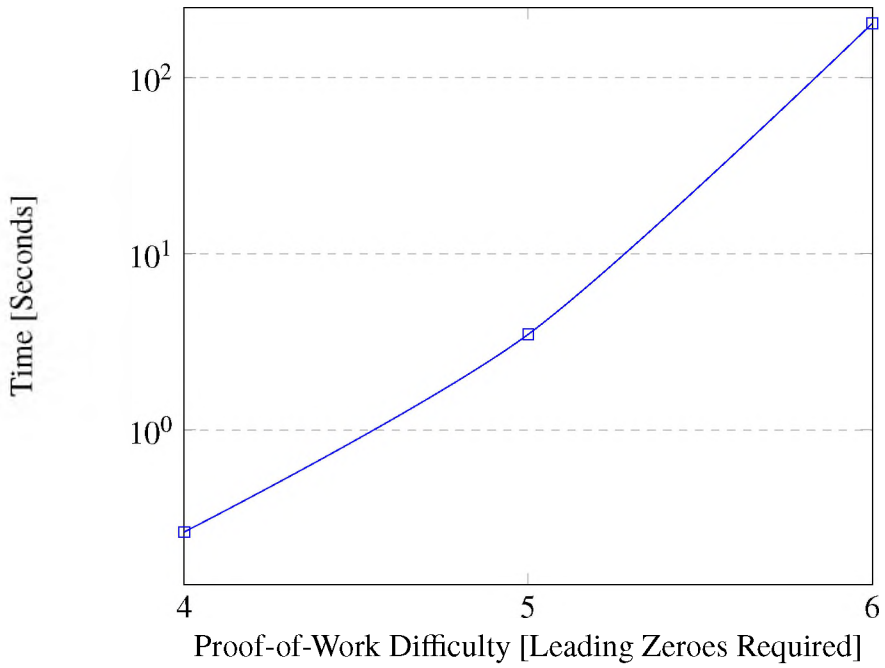


Figure 5.11. Speed per Proof-of-Work Difficulty

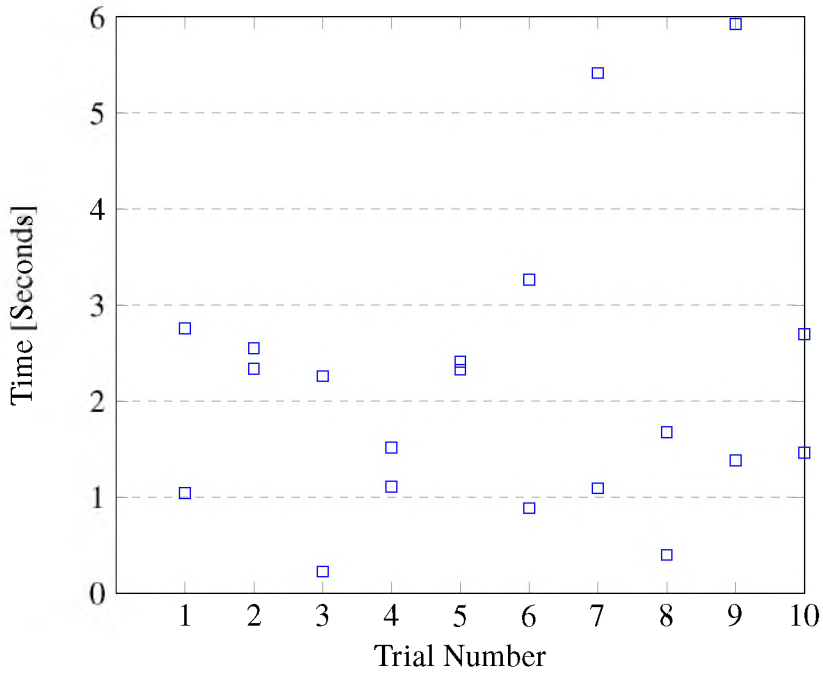


Figure 5.12. Certificate Time

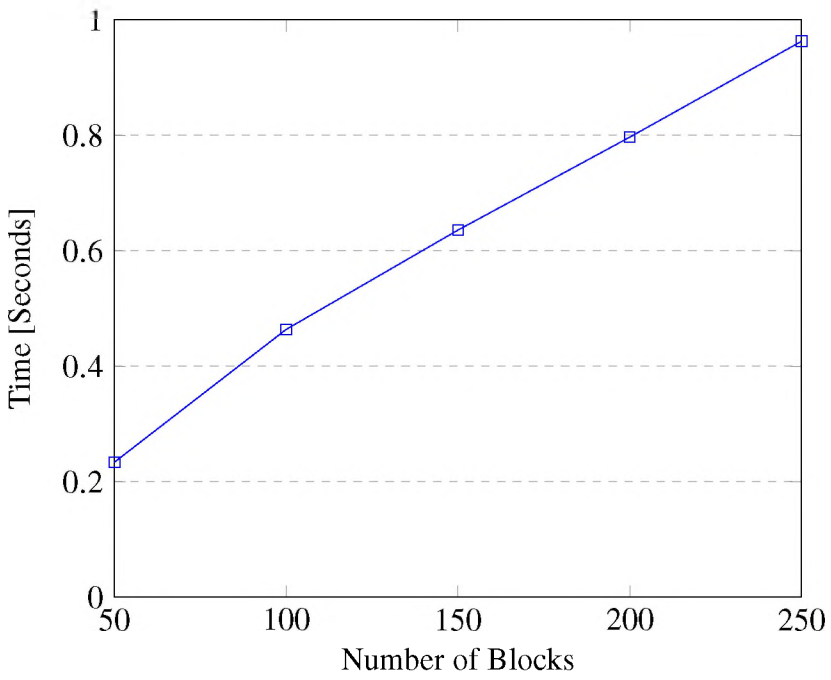


Figure 5.13. Joining Speed per Number of Blocks

for a real-time system such as a VANET. In the proposed system, there is no benefit from being the vehicle to produce the next block. Thus, the cost of using PoW as a consensus mechanism has no advantage.

6. EFFICIENT BLOCKCHAIN AUTHENTICATION SCHEME FOR VANETS

To solve the issues faced with using PoW as a consensus mechanism and the scalability issues of the previously proposed Join Platoon protocol some alterations were made. Due to the low number of nodes in a platoon compared to the network of Bitcoin, a consensus mechanism using the Schnorr Multi-Signature coupled with an algorithm designed to reach consensus in the presence of partially synchronous byzantine faults is proposed to solve these issues. A truncated history is used to join new platoons instead of the entire blockchain. Both of these changes are proven secure and show to reduce the resource requirements of the protocols.

6.1. PROPOSED SCHEME

This section presents the protocols used by participants in the VANET. In particular, vehicle registration, platoon join, block creation, intra platoon communication, and platoon leave protocols are presented. The only protocol where the CA or any infrastructure component is present is the vehicle registration protocol.

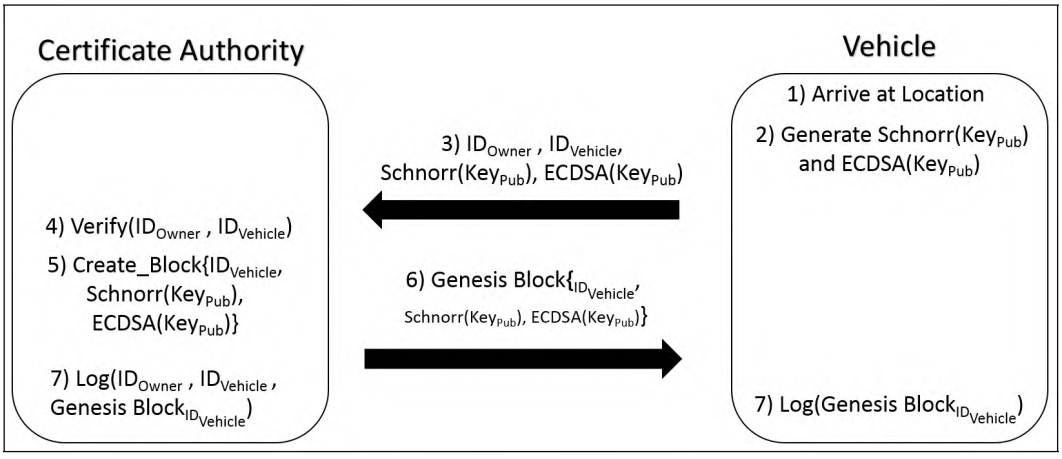


Figure 6.1. Vehicle Registration Protocol

6.1.1. Vehicle Registration Protocol. The proposed system uses private blockchains. This means that all users must be registered with a CA to participate. This is a standard requirement for VANETs since it is what you will currently find in transportation infrastructures in most countries. The CA is charged with inspecting the vehicle, requesting any fees or taxes, and creating a certification for the vehicle. This certification allows the vehicle to begin participating in the VANET and take advantage of the cost-saving opportunities it provides and can be considered the genesis block of the vehicle's blockchain. The Vehicle Registration Protocol is outlined in Figure 6.1 and works as follows.

1. First, the vehicle arrives at a registration station owned by the CA. This will take the form of a Department of Motor Vehicle office or Licensed Mechanic who can verify the physical properties of the vehicle.

2. The vehicle then generates an ECDSA Public/Private Key Pair. This key pair is used to sign transactions containing evaluations of other vehicle's actions while a part of a platoon. The vehicle will also generate a Schnorr Multi-Signature Public/Private Key Pair. This signature is used during the creation of new certification blocks for vehicles leaving the platoon and during the intra platoon communication protocol.

3. The vehicle sends the owner's identification, its identification, and both public keys to the CA.

4. The CA will verify the identification, checking that both the vehicle and owner are valid.

5. The CA will then create a genesis block containing the vehicle's id, both public keys, and a signed certificate from the CA.

6. Both the CA and the vehicle will log this information. The CA keeps it so that it can penalize the vehicle and vehicle owner as needed.

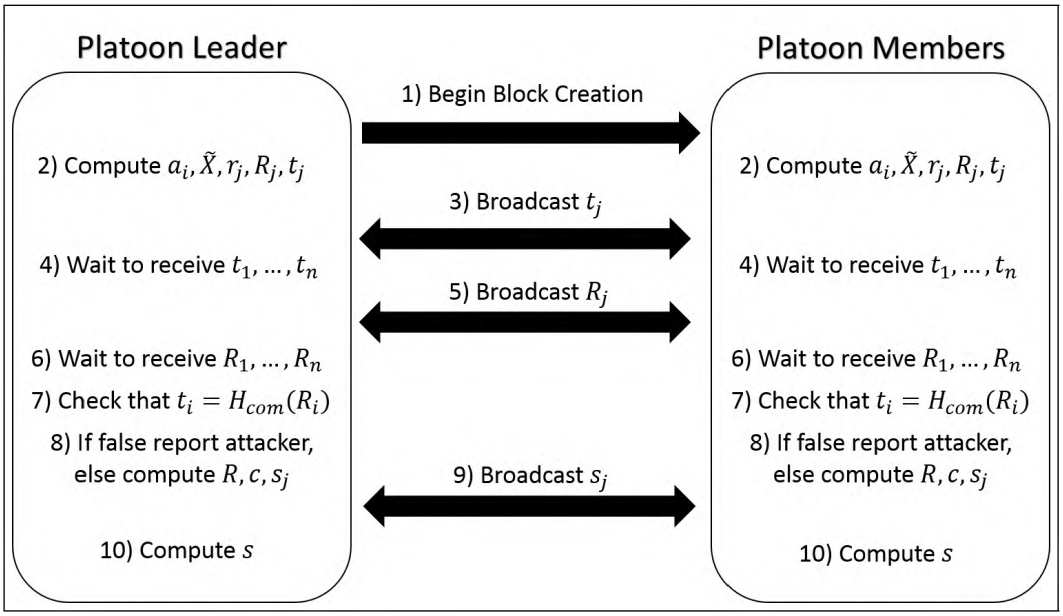


Figure 6.2. Block Creation Protocol

Once this protocol is complete, the vehicle is free to join its first platoon and benefit from the cost-saving applications of the VANET. It will not need to register again unless it gets kicked out of the platoon for possessing a "bad" block.

In the proposed system, a "good" block simply denotes a vehicle that has behaved correctly while a "bad" block indicates a vehicle that has not. The definition of correctness is explained in Section 6.2.

6.1.2. Block Creation Protocol. Blocks are generated whenever there is a change in the state of the platoon. This protocol is adapted from the Schnorr Multi-Signature scheme and applied to a platoon. It must be carried out and a platoon signature created for a vehicle when they want to leave the platoon to be allowed to join another platoon. If a vehicle does not possess a valid block when they leave, they will not be able to join any future platoons. The block generated by the protocol serves as certification showing that the vehicle behaved correctly while a part of the platoon according to all cyber and physical actions performed by the vehicle. An outline of the protocol is seen in Figure 6.2 and works as follows.

1. The platoon leader indicates to the platoon that they will begin the block creation protocol via broadcasting a signed message.

2. For $i \in 1, ..., n$, every vehicle in the platoon then computes $a_i = H_{agg}(L, X_i)$. The aggregated public key for the platoon is then $\tilde{X} = \prod_{i=1}^{n} X_i^{a_1}$. Each platoon member also generates a random $r_j \leftarrow \mathbb{Z}_p$, computes $R_j = g^{r_j}$, and $t_j = H_{com}(R_j)$.

3. Each platoon member broadcasts t_j to all other members of the platoon.

4. The platoon waits until it receives all t from every platoon member.

5. Once every platoon member gets $t_2, ..., t_n$ from the other platoon members, it broadcasts R_j to the entire platoon.

6. The platoon waits until it receives all R from every platoon member.

7. Once it gets $R_2, ..., R_n$ it checks that $t_i = H_{com}(R_i)$ for all $i \in 2, ..., n$.

8. If it is not true, the platoon aborts the computation and creates a transaction evaluating the faulty vehicle. Otherwise, every vehicle in the platoon computes $R = \prod_{i=1}^{n} R_i$, $c = H_{sig}(\tilde{X}, R, m)$,, $s_j = r_j + ca_j x_j mod p$.

9. Every vehicle in the platoon sends s_1 to all other platoon members.

10. Once the all vehicles in the platoon receive $s_2, ...s_n$ from the platoon members, it computes $s = \sum_{i=1}^{n} s_1 mod p$ and the signature for the message is $\sigma = (R, s)$.

To ensure that consensus on the values broadcast at steps 3, 5, and 9, Algorithm 2 is applied from (Dwork *et al.*, 1988) which is used to reach consensus in the face of Byzantine faults under partially synchronous communication and synchronous processors when authentication is present. In the proposed protocol, the digital signature is used when sending messages. Additionally, a secure GPS is located within each car that is used for navigation. It is used to synchronize the processors of all the vehicles in the platoon.

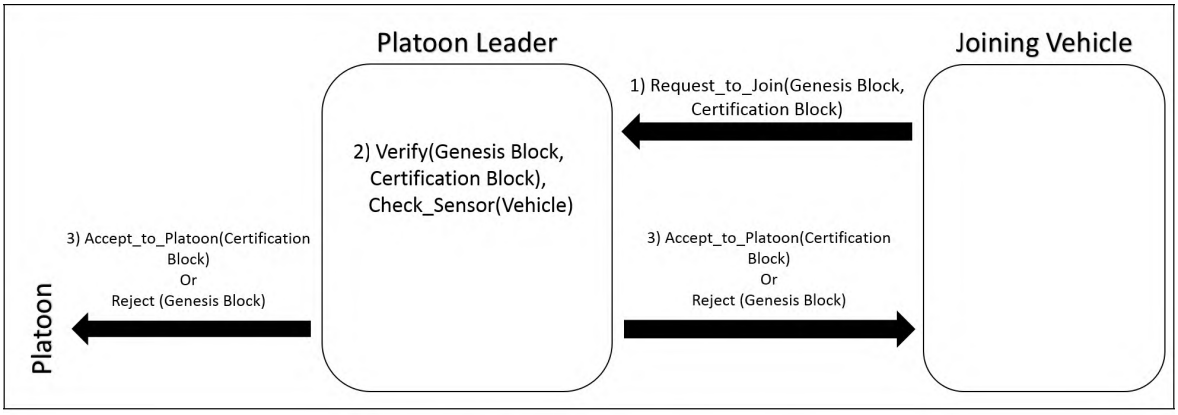


Figure 6.3. Platoon Join Protocol

6.1.3. Platoon Join Protocol. Whenever a vehicle attempts to join a platoon, its last certification block must be verified by the platoon it is attempting to join. When a vehicle joins a platoon, its secure GPS reports the total distance it has traveled. The platoon is trusting that the previous platoon behaved correctly and gave the vehicle the appropriate designation of "good" or "bad". The correctness of this assumption is proven in Section 6.2. To understand how this protocol works, an outline is given in Figure 6.3 and described below.

1. The vehicles sends a request to join message with includes its genesis block and the certification block from its last platoon.

2. The platoon will verify that the vehicle is valid by validating the genesis block and the certification from the CA. Additionally, it will verify that the vehicle is physically present within the sensor range of the platoon.

3. If it is valid, it will continue to validate the certification block as follows: given a multi-set of public keys $L = X_1, ..., X_n$, a message m, and a signature $\sigma = (R, s)$, the new platoon compute $a_i = H_{agg}(L, X_i)$ for each $i \in 1, ..., n$, $\tilde{X} = \prod_{i=1}^{n} X_i^{a_i}$, $c = H_{sig}(\tilde{X}, R, m)$. It will

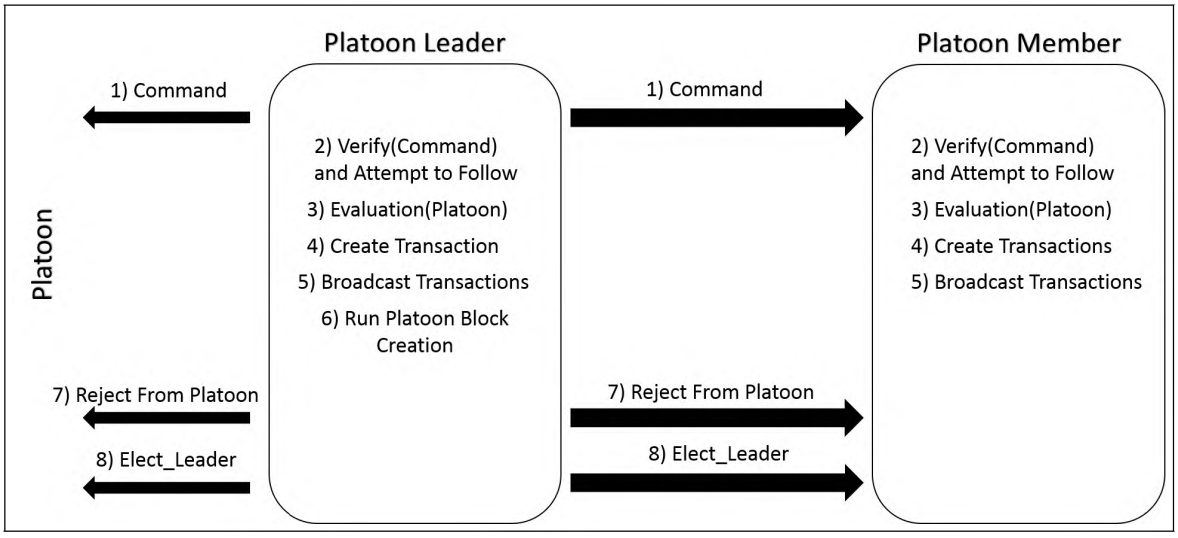


Figure 6.4. Intra Platoon Communication Protocol

then accept the certification if $G^s = R\prod_{i=1}^{n} X_i^{a_i c} = RX^c$. Otherwise, it will simply reject the vehicle from participating in the platoon. The platoon leader will then tell the platoon whether or not to accept the requesting vehicle.

6.1.4. Intra Platoon Communication Protocol. Every time a command is issued by the platoon leader, this protocol is run to disseminate transactions and detect faults by vehicles within the platoon. A brief outline of this protocol is given in Figure 6.4 and described below.

1. The platoon leader issuing a command to the platoon.

2. Every vehicle within the platoon then receives the command, verifies that it is from the platoon leader, and attempts to follow the command assuming that the result will not end in a bad state.

3. As the platoon members are following the command, they monitor one another according to the invariants of the system. Once the platoon maneuver is complete, every vehicle creates transactions for every other vehicle in the platoon.

4. Every vehicle in the platoon broadcasts its transactions to the other vehicles within the platoon.

5. The platoon will run the block creation protocol to reach a consensus on the actions of the vehicles within the platoon.

6. If any vehicles behaved inappropriately during the maneuver, they are deemed untrustworthy and kicked from the platoon.

7. If the platoon leader is kicked, a new platoon leader is elected from the remaining vehicles.

6.1.5. Platoon Leave Protocol. To leave the platoon, a vehicle must receive a certification block from the platoon. Otherwise, it will not be allowed to join any future platoons. A brief description of this protocol can be seen is Figure 6.5. Once the vehicle leaves the platoon, it will use the last signed leave-platoon request to join the next platoon. An outline of the protocol is given below.

1. The vehicle broadcasts a message to the entire platoon that it is leaving the platoon.

2. The platoon leader then initiates the platoon block creation protocol to sign a message indicating that the platoon received the leave platoon request.

3. The platoon leader waits until the protocol is complete.

4. Once the platoon leader receives the signature, it sends an acknowledge message to the leaving vehicle.

5. The platoon leader then waits while the leaving vehicle leaves the platoon and is a safe distance away so that it cannot interfere with the platoon maneuvers.

6. The platoon leader sends the signed message to the leaving vehicle.

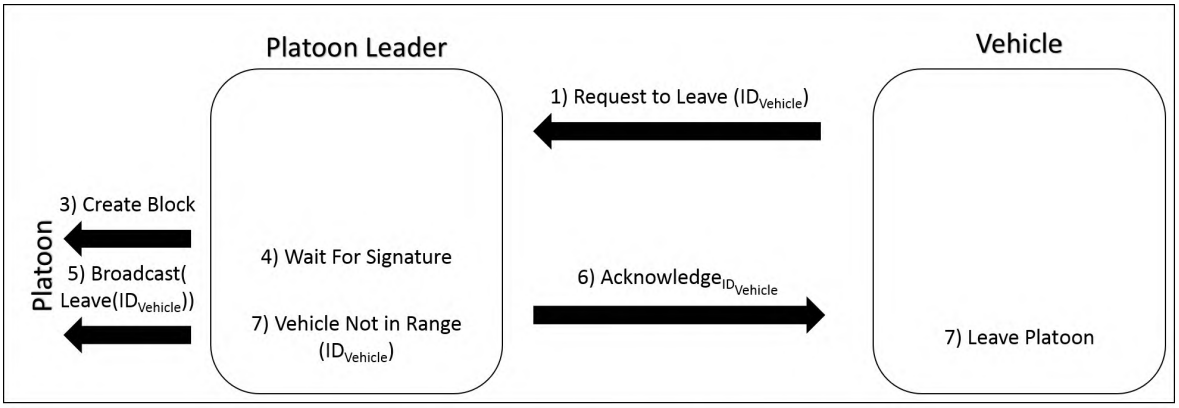


Figure 6.5. Platoon Leave Protocol

6.2. SECURITY PROOF

To formally prove the security of the proposed scheme, several different theorems about this work are proven. First, a list of seven different assumptions that are assumed in the model is given. After those are presented, some basic definitions of the proposed system are laid-out. Lastly, several theorems that show the strength of the protocols are proven.

Assumption 1. *The CA that generates the certificates for the vehicles is a trusted entity and will not reveal any information about a vehicle* $\boldsymbol{V}$ *to an attacker* $\boldsymbol{A}$.

Assumption 2. *A vehicle* $\boldsymbol{V}$ *will not reveal its private signing information.*

Assumption 3. *There are a limited number of attackers* $\boldsymbol{A}$ *where* $N > 3A + 1$. N *is the number of vehicles in a platoon. This assumption is based on previous work discussing the maximum number of attackers to reach consensus under partial synchronicity in the face of the byzantine faults with authentication. This number of attacks is based on (Dwork* et al., *1988).*

Assumption 4. *There is a tamper-proof GPS.*

Assumption 5. *There is a bounded distance* D_B *that a vehicle can drive without receiving a new certification block before it will no longer be allowed to join a platoon.*

Assumption 6. *A vehicle **V** can only be a part of one platoon at a time.*

Assumption 7. *For every action that is in the blockchain, there must be a certifiable action, either cyber or physical, that is evaluated by the platoon.*

Throughout this section, correctness refers to a vehicle's actions both in the cyber and physical domains of the vehicle. This works' description of correctness follows directly from Assumption 7.

Definition 1. A vehicle is behaving correctly if it passes the evaluation and certification of its actions by other vehicles within the platoon that will be stored in its blockchain.

The description of correctness is purposefully left vague to avoid requiring in-depth and lengthy proofs. Based on the aforementioned assumptions, the proposed scheme can be described with the following definitions. The description of a platoon in a VANET, the requirements to join a platoon is given, and the result of leaving a platoon is given.

Definition 2. Platoon **P** is a group of **N** vehicles that drive in the proximity of one another and cooperate for some particular application where **N** is bounded. The vehicles within the platoon carry out the proposed communication protocols as needed when vehicles join or leave and whenever a physical action is carried out by the platoon.

Definition 3. To join platoon **P**, vehicle **V** must possess a valid certification block C_V, be moving in the same direction as the platoon, and by physically sensible by the platoon.

Definition 4. Whenever a vehicle **V** leaves platoon **P**, it will receive a valid certification block C_V that will denote whether it is behaving correctly based on Definition 1.

Now that the proposed scheme and protocols have been defined in terms of the system and the base assumptions of the been discussed, the security proof begins by proving several base theorems about the proposed system.

Theorem 6.2.1. *A vehicle **V** can only be a part of one block creation event at a time.*

Proof. Theorem 6.2.1 is proven by contradiction. Assume that **V** created two blocks at the same time. Blocks are generated by joining a platoon and subsequently leaving the same platoon (Definition 4) or by participating in a platoon maneuver as denoted in Section 6.1. Thus, **V** would have had to join two platoons and participated in a maneuver within both or subsequently left both. To join a platoon, a vehicle must have a valid certification block that shows that they behave correctly and must be physically a part of that platoon (Definition 3). Since **V** cannot be two places at once, it cannot be a part of two separate platoons. Thus, it cannot create two blocks during a single period and Theorem 1 is proven. □

Theorem 6.2.2. *A vehicle **V** will not be able to change the contents of their certification block C_V when they are not a part of a platoon.*

Proof. Theorem 6.2.2 is proven by contradiction. Assume that vehicle **V** was able to change their certification block C_V when they were not a part of a platoon. To change C_V two cases could have happened. First, **V** could have reverted to a previous certification block. Secondly, **V** could have changed the contents of C_V.

Case 1: Assume the first case where **V** reverted to a previous certification block. To begin, assume that **V** has traveled more than the bounded distance D_B since receiving the previous certification block. In this case, the certification block would contain GPS reading D_O. No GPS can be falsified due to Assumption 4. Given the current GPS reading D_C, $D_C - D_O > D_B$, thus it will not be able to join a platoon due to Assumption 5

Secondly, assume that **V** has traveled less than D_B since receiving the previous certification block. Since it has traveled less than D_B, the vehicle has either not moved since leaving the last platoon or it could be moving. Given its current GPS reading D_C and the GPS reading contained within the certification block D_O that $D_C - D_O < D_B$. If it was not moving **V** would be unable to participate in a platoon since the traffic would be moving and a **V** is required to be moving with the platoon and be physically sensible by the platoon to join (Definition 5.8). Thus, **V** would be unable to use C_V to participate in any platoon when it cannot join due to the physical constraints of the platoon-join requirements. Thus,

V would be unable to use the previous certification block to participate due to Assumption 5. If **V** was moving, then it will have a period before it travels D_B until the previous certification block is invalid. Thus, **V** will eventually be caught using a false certification block and not be allowed to join a platoon.

Case 2: Assume the second case where **V** changed the contents of C_V. If **V** changed the contents of C_V then **V** would have to possess the private signing information of all vehicles in the prior platoon that was used to create C_V or a fork was created in the platoon's blockchain. However, a vehicle will not reveal it's private signing information due to Assumption 2. Furthermore, it cannot create two blocks simultaneously due to Theorem 6.2.1. Thus, they could not have changed the contents of C_V.

Since **V** could not use a previous certification block and could not have changed the contents of C_V, the assumption that **V** was able to change their certification block C_V is incorrect. This proves theorem 6.2.2. □

Theorem 6.2.3. *A vehicle V will always have a valid certification block C_V whenever it attempts to join a platoon.*

Proof. Theorem 6.2.3 will be proven by induction. First, the proof begins with the base case. Let us prove that a vehicle **V** will have a valid certification block C_V when it attempts to join its first platoon, P_1. In this case, its last certification block will have been created by the CA. Due to Assumption 1, this certification block is valid. Additionally, based on Theorem 6.2.2, **V** was unable to change C_V. Since the vehicle has received a valid certification block from the CA and was unable to change it, **V** will join P_1 with a vehicle certification block. Thus, **V** will be allowed to join the platoon if C_V says **V** has behaved correctly or will be denied if it says **V** has behaved incorrectly.

Next, the inductive case is proven. Assume that **V** has a valid certification block C_V holds whenever **V** joined P_N. Let us prove that **V** has also has a valid certification block C_V when **V** tries to join P_{N+1}. Before **c**an join P_{N+1}, it must leave P_N due to theorem 6.2.1 and Assumption 6. When **V** leaves the platoon P_N, it will create a secure certification block for

V. If there are **X** vehicles in the platoon not including **V**, then it follows that there are only up to **Y** attackers where $Y < \frac{X}{3}$ based on Assumption 3. In the block creation algorithm, vehicles exchange evaluations of other vehicles with one-another and the majority score for any vehicle is taken as the cumulative value. Thus, when the cumulative trust value for a vehicle is calculated, the output will follow the answer created by the honest portion of the platoon since the number of honest vehicle **H** since $H = X - Y$ thus $H >= \frac{2X}{3}$. This results in **V** receiving a valid certification block when they leave the platoon.

Due to theorem 6.2.2, **V** cannot alter C_V after it leaves P_N. Thus, **V** will use C_V to join P_N+1. When **V** attempts joining P_N+1 using C_V it will be allowed to join the platoon if C_V says **V** has behaved correctly or will be denied if it says **V** has behaved incorrectly.

Since theorem 6.2.3 holds when **V** joins P1 and it was shown that if C_V is valid when **V** joins P_N then C_V will be valid when it attempts to join P_N+1, theorem 6.2.3 is proven. □

Theorem 6.2.4. *The "cyber-physical" blocks that are created in the form of certification blocks encapsulate both the cyber and physical domains.*

Proof. Theorem 6.2.4 is proven by contradiction. Assume that the certification blocks do not encapsulate both the cyber and physical domains. This means that it can either not encapsulate the cyber system or not encapsulate the physical system. Assumption 7 says that for every transaction in the blockchain that evaluates a vehicle **V**, it will be the cyber representation of some action by **V**. These actions can fall into two categories: cyber and physical. Cyber actions are the actions that **V** takes as part of the system that does not result in direct physical action. This includes evaluating other vehicles, making actions of other vehicles, and simply replying to messages from the platoon within a specified time-bound. Physical actions are any action that **V** takes that result in physical action by V. These include braking, accelerating, and turning. Thus, by Assumption 7, the blockchain will include both cyber and physical actions since they are both verifiable actions. □

Table 6.1. Table of Symbols for MSDND Proof

Symbol	Definition
CP	The consensus protocol of the platoon
LB_2	The local blockchain of vehicle 2
VC_1/VC_2	The vehicle controller of vehicle 1 or vehicle 2 respectively
VO_2	The physical vehicle operations of vehicle 2
CC_1/CC_2	The cyber communications of vehicle 1 or vehicle 2 respectively
S_1	The sensor unit of vehicle 1
FP_2	The future platoon of vehicle 2

Now that some basic properties of the proposed protocol have been proven, some theorems describing the benefit of the proposed approach to evaluating both the physical and cyber portions of the system, instead of one or the other, are presented and proven. These proofs use MSDND to show the security of the approach. In MSDND, $IBT_{1,2}Val$ is a macro used to describe the information flow from one entity to another in a system model (Palaniswamy and McMillin, 2018). It means that entity 2 reported to entity 1 the value Val is true and entity 1 believes entity 2.

In a physical-only blockchain, there is no information flow path from VO_2 to S_1 as seen in Figure 6.6. A table of shorthand notations used for this and the following proofs can be seen in Table 6.1. Let φ_1 be the statement "Vehicle 2 is maneuvering correctly". Let φ_2 be the statement "Vehicle 2 is communicating correctly with other vehicles". The definition of correctness comes from Definition 1. Either φ_1 or $\neg\varphi_1$ must be true at all times. Similarly, either φ_2 or $\neg\varphi_2$ must be true at all times. Finally, $\varphi = \varphi_1 \wedge \varphi_2$ means that the vehicle is behaving correctly. In this system, repeated evaluations of other vehicles that are noted in the local blockchain are used by future platoons to evaluate the trustworthiness of a vehicle. Thus, there is an information flow path from the consensus protocol of the platoon to the local blockchain of a vehicle and a path from the local blockchain of a vehicle to any future platoons of that vehicle.

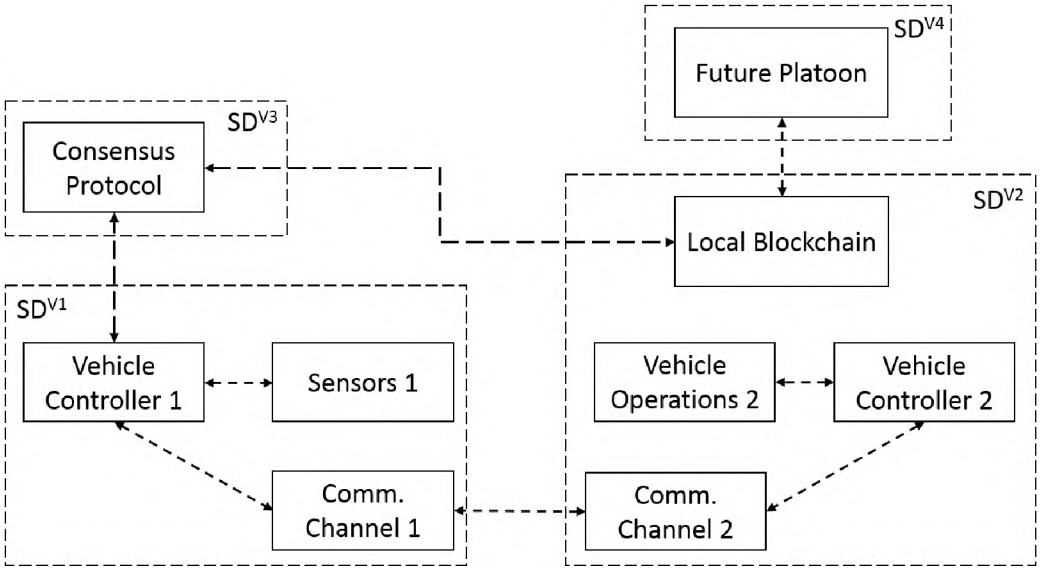


Figure 6.6. Cyber Only Blockchain Information Flow Diagram

Theorem 6.2.5. *A cyber-only blockchain is not MSDND secure under an attack on the cyber communications of a vehicle.*

Proof. Assume that in a cyber-only blockchain, some function f exists to determine whether φ_2 is true or false that is owned by CP. This follows from Assumption 7. In the model, assume that CP will always be honest due to the bounded number of attackers in Assumption 3.

1. $\neg\varphi_2 = true$; Vehicle 2 is not sending correct cyber communications.

2. $w \vDash V_{\varphi_2}^{VC_2}(w) = true$; VC_2 observes that they are communicating correctly.

3. $IBT_{CC_2,VC_2}\varphi_2$; VC_2 lies to CC_2 and tells it that the cyber communicating are correct.

4. $w \vDash V_{\varphi_2}^{CC_2}(w) = true$; CC_2 observes that the cyber communications from V_2 are correct.

5. $IBT_{CC_1,CC_2}\varphi_2$; CC_2 sends the correct cyber communications to CC_1.

6. $w \vDash V_{\varphi_2}^{CC_1}(w) = true$; CC_1 observes that V_2 sent the correct cyber communications.

7. $IBT_{VC_1,CC_1}\varphi_2$; CC_1 tells VC_1 that V_2 sent the correct cyber communications.

8. $w \vDash V_{\varphi_2}^{VC_1}(w) = true$; VC_1 observes that V_2 sent62789-= the correct cyber communications.

9. $IBT_{CP,VC_1}\varphi_2$; VC_1 tells CP that V_2 sent the correct cyber communications.

10. $w \vDash V_{\varphi_2}^{CP}(w) = true$; CP observes that V_2 sent the correct cyber communications.

11. $\neg\varphi_2 \Rightarrow \neg f$; since $\neg\varphi_2 = true$ then function $\neg f = true$.

12. $IBT_{CP,f}\neg\varphi_2$; f tells CP that V_2 sent the incorrect cyber communications.

13. $w \vDash V_{\neg\varphi_2}^{CP}(w) = true$; CP has now deduced that V_2 sent the incorrect cyber communications.

14. $w \vDash V_{\neg\varphi_2}^{CP}(w) = true \implies w \vDash V_{\neg\varphi_2}^{FP_2}(w) = true$; since a valuation function exists at CP to evaluate φ_2 it follows that there also exists a valuation function at FP_2 to evaluate φ_2.

15. $IBT_{LB_2,CP}\neg\varphi_2$; CP tells LB_2 that V_2 sent the incorrect cyber communications.

16. $w \vDash V_{\neg\varphi_2}^{LB_2}(w) = true$; LB_2 observes that V_2 sent the incorrect cyber communications.

17. $IBT_{FP_2,LB_2}\neg\varphi_2$; LB_2 tells FP_2 that V_2 sent the incorrect cyber communications.

18. $w \vDash V_{\neg\varphi_2}^{FP_2}(w) = true$; FP_2 observes that V_2 sent the incorrect cyber communications.

19. $\neg$MSDND(ES): $\exists w \in W \vdash [(\varphi_2 \oplus \neg\varphi_2)] \wedge [w \models (\exists V_{\varphi_2}^{FP_2}(w))]$

FP_2 has a valuation of φ_2. Therefore, the cyber action readings are not MSDND secure to FP_2. This means FP_2 will know the truth behind whether V_2 was behaving correctly on a cyber level in prior platoons. □

Theorem 6.2.6. *A cyber-only blockchain is MSDND secure under an attack on the physical maneuvers of a vehicle.*

Proof. 1. $\neg\varphi_1 = true$; The vehicle 2 is not maneuvering correctly.

2. $w \vDash V_{\varphi_1}^{VC_2}(w) = false$; VC_2 observes that vehicle 2 is not maneuvering correctly.

3. $IBT_{CC_2,VC_2}\varphi_1$; VC_2 lies to CC_2 and tells it that vehicle 2 is maneuvering correctly.

4. $w \vDash V_{\varphi_1}^{CC_2}(w) = true$; CC_2 observes that the vehicle 2 is maneuvering correctly.

5. $IBT_{CC_1,CC_2}\varphi_1$; CC_2 tells CC_1 that vehicle 2 is maneuvering correctly

6. $w \vDash V_{\varphi_1}^{CC_1}(w) = true$; CC_1 observes that vehicle 2 is maneuvering correctly.

7. $IBT_{VC_1,CC_1}\varphi_1$; CC_1 tells VC_1 that vehicle 2 is maneuvering correctly.

8. $w \vDash V_{\varphi_1}^{VC_1}(w) = true$; VC_1 observes that vehicle 2 is maneuvering correctly.

9. $IBT_{CP,VC_1}\varphi_1$; VC_1 tells CP that vehicle 2 is maneuvering correctly.

10. $w \vDash V_{\varphi_1}^{CP}(w) = true$; CP observes that vehicle 2 is maneuvering correctly.

11. $IBT_{LB_2,CP}\varphi_1$; CP tells LB_2 that vehicle 2 is maneuvering correctly.

12. $w \vDash V_{\varphi_1}^{LB_2}(w) = true$; LB_2 observes that vehicle 2 is maneuvering correctly.

13. $IBT_{FP_2,LB_2}\varphi_1$; LB_2 tells FP_2 that vehicle 2 is maneuvering correctly.

14. $w \vDash V_{\varphi_1}^{FP_2}(w) = true$; FP_2 observes that vehicle 2 is maneuvering correctly.

15. MSDND(ES): $\exists w \in W \vdash [(\varphi_1 \oplus \neg\varphi_1)] \wedge [w \models (\nexists V_{\varphi_1}^{FP}(w))]$

This proof follows similarly to the last except for the valuation function f. Thus, FP_2 believes the false physical action reading reported by LB_2. Therefore, the physical action readings are MSDND secure to FP_2. This means FP_2 will not know the truth behind whether V_2 was behaving correctly on a physical level in prior platoons. □

Lemma 6.2.6.1. *Since CP is the only entity with an information flow to LB_2 and LB_2 is the only entity with information flow to FP_2 it follows that if there exists a world such that* $w \vDash V_{\neg\varphi_2}^{CP}(w) = true \implies w \vDash V_{\neg\varphi_2}^{FP_2}(w) = true$.

Proof. This lemma follows from the fact that LB_2 cannot change its history since it is a read-only ledger belonging to V_2. Thus, since LB_2 cannot be malicious, it follows that it will pass on the same information that it receives from CP. Thus, if CP has a valuation function that can evaluate the truth of φ_2, then so does FP_2. □

Lemma 6.2.6.2. *FP_2 will receive the correct information regardless of whether V_1 is malicious or not.*

Proof. Since the number of attackers in bounded by Assumption 3, CP will always have a valuation function that satisfies both the adapted IC1 and IC2. Thus, it will reach the correct valuation regardless of the presence of a bounded number of malicious vehicles. □

Corollary 6.2.6.1. *It follows that in a cyber-only blockchain, that the physical actions of vehicle 2 can be successfully altered while the cyber actions of vehicle 2 cannot be successfully altered to deceive the future platoon of vehicle 2.*

Proof. Both state variables φ_1 and φ_2 are independent of one another. This means that a vehicle can behave incorrectly on either the cyber level or physical level without forcing incorrect actions at the other level. □

This proof shows the inherent weakness of a cyber-only blockchain applied to a cyber-physical system. If a future platoon can be deceived about what a vehicle's actions were in past platoons then it is insecure since it cannot fully trust the joining vehicle. Now, a proof is presented proving the security of a physical-only blockchain in the proposed architecture. In a physical-only blockchain, there is no information flow path from CC_2 to CC_1 as seen in Figure 6.7.

Theorem 6.2.7. *A physical-only blockchain is not MSDND secure under an attack on the physical level of the system.*

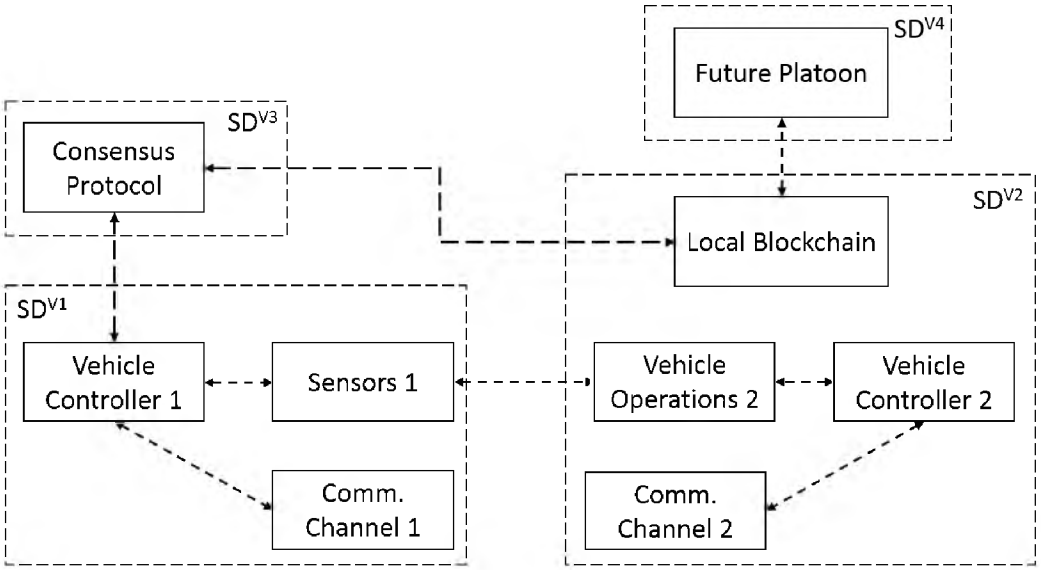


Figure 6.7. Physical Only Blockchain Information Flow Diagram

Proof. Assume that in a cyber-only blockchain, some function f exists to determine whether φ_1 is true or false that is owned by CP. This follows from Assumption 7. In the model, it is assumed that CP will always be honest due to the bounded number of attackers in Assumption 3.

1. $\neg\varphi_1 = true$; The V_2 is not maneuvering correctly.

2. $w \vDash V_{\varphi_1}^{VC_2}(w) = false$; VC_2 observes that V_2 is not maneuvering correctly.

3. $IBT_{VO_2,VC_2}\varphi_1$; VO_2 lies to VO_2 and tells it that V_2 is maneuvering correctly.

4. $w \vDash V_{\varphi_1}^{VO_2}(w) = true$; VO_2 observes that the V_2 is maneuvering correctly.

5. $IBT_{S_1,VO_2}\varphi_1$; VO_2 tells S_1 that V_2 is maneuvering correctly.

6. $w \vDash V_{\varphi_1}^{S_1}(w) = true$; S_1 observes that V_2 is maneuvering correctly.

7. $IBT_{VC_1,S_1}\varphi_1$; S_1 tells VC_1 that V_2 is maneuvering correctly.

8. $w \vDash V_{\varphi_1}^{VC_1}(w) = true$; VC_1 observes that V_2 is maneuvering correctly.

9. $IBT_{CP,VC_1}\varphi_1$; VC_1 tells CP that V_2 is maneuvering correctly.

10. $w \vDash V_{\varphi_1}^{CP}(w) = true$; CP observes that V_2 is maneuvering correctly.

11. $\neg\varphi_1 \Rightarrow \neg f$; since $\neg\varphi_1 = true$ then function $\neg f = true$.

12. $IBT_{CP,f}\neg\varphi_1$; f tells CP that V_2 is maneuvering incorrectly.

13. $w \vDash V_{\neg\varphi_1}^{CP}(w) = true$; CP has now deduced that V_2 is maneuvering correctly.

14. $w \vDash V_{\neg\varphi_1}^{CP}(w) = true \implies w \vDash V_{\neg\varphi_1}^{FP_2}(w) = true$; since a valuation function exists at CP to evaluate φ_1 it follows that there also exists a valuation function at FP_2 to evaluate φ_1.

15. $IBT_{LB_2,CP}\varphi_1$; CP tells LB_2 that V_2 is maneuvering correctly.

16. $w \vDash V_{\varphi_1}^{LB_2}(w) = true$; LB_2 observes that V_2 is maneuvering correctly.

17. $IBT_{FP_2,LB_2}\varphi_1$; LB_2 tells FP_2 that V_2 is maneuvering correctly.

18. $w \vDash V_{\varphi_1}^{FP_2}(w) = true$; FP_2 observes that V_2 is maneuvering correctly.

19. $\neg$ MSDND(ES): $\exists w \in W \vdash [(\varphi_1 \oplus \neg\varphi_1)] \wedge [w \models (\exists V_{\varphi_1}^{FP}(w))]$

FP_2 has a valuation of φ_2. Therefore, the physical action readings are not MSDND secure to FP_2. This means FP_2 will know the truth behind whether V_2 was behaving correctly on a physical level in prior platoons. □

Theorem 6.2.8. *A physical-only blockchain is MSDND secure under an attack on the cyber-level of the system.*

Proof. 1. $\neg\varphi_2 = true$;Vehicle 2 is not sending the correct cyber communications

2. $w \vDash V_{\varphi_2}^{VC_2}(w) = true$; VC_2 observes that they are communicating correctly.

3. $IBT_{VO_2,VC_2}\varphi_2$; VC_2 lies to VO_2 and tells it that the cyber communicating are correct.

4. $w \vDash V_{\varphi_2}^{VO_2}(w) = true$; VO_2 observes that the cyber communications from vehicle 2 are correct.

5. $IBT_{S_1,VO_2}\varphi_2$; VO_2 sends the correct cyber communications to S_1.

6. $w \vDash V_{\varphi_2}^{S_1}(w) = true$; S_1 observes that vehicle 2 sent the correct cyber communications.

7. $IBT_{VC_1,S_1}\varphi_2$; S_1 tells VC_1 that vehicle 2 sent the correct cyber communications.

8. $w \vDash V_{\varphi_2}^{S_1}(w) = true$; S_1 observes that vehicle 2 sent the correct cyber communications.

9. $IBT_{CP,VC_1}\varphi_2$; VC_1 tells CP that vehicle 2 sent the correct cyber communications.

10. $w \vDash V_{\varphi_2}^{CP}(w) = true$; CP observes that vehicle 2 sent the correct cyber communications.

11. $IBT_{LB_2,CP}\varphi_2$; CP tells LB_2 that vehicle 2 sent the correct cyber communications.

12. $w \vDash V_{\varphi_2}^{LB_2}(w) = true$; LB_2 observes that vehicle 2 sent the correct cyber communications.

13. $IBT_{FP_2,LB_2}\varphi_2$; LB_2 tells FP_2 that vehicle 2 sent the correct cyber communications.

14. $w \vDash V_{\varphi_2}^{FP_2}(w) = true$; FP_2 observes that vehicle 2 sent the correct cyber communications.

15. MSDND(ES): $\exists w \in W \vdash [(\varphi_2 \oplus \neg\varphi_2)] \wedge [w \models (\nexists V_{\varphi_2}^{FP_2}(w))]$

This proof follows similarly to the last except for the valuation function f. FP_2 does not have a valuation of φ_2. Therefore, the cyber action readings are MSDND secure to FP_2. This means FP_2 will not know the truth behind whether vehicle 2 was behaving correctly on a cyber level in prior platoons. □

Lemma 6.2.8.1. *Since CP is the only entity with an information flow to LB_2 and LB_2 is the only entity with information flow to FP_2 it follows that if there exists a world such that $w \vDash V_{\neg\varphi_1}^{CP}(w) = true \implies w \vDash V_{\neg\varphi_1}^{FP_2}(w) = true$.*

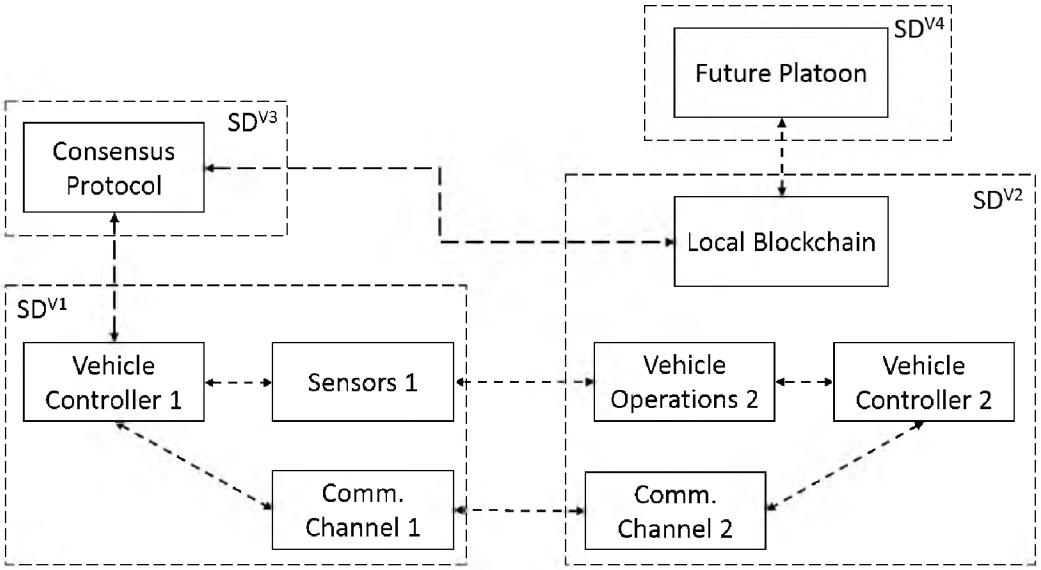


Figure 6.8. Cyber Physical Blockchain Information Flow Diagram

Corollary 6.2.8.1. *It follows that in a physical-only blockchain, that the cyber actions of vehicle 2 can be successfully altered while the physical actions of vehicle 2 cannot be successfully altered to deceive the future platoon of vehicle 2.*

Proof. See corollary 6.2.6.1 for a similar proof. □

Theorem 6.2.9. *A cyber-physical blockchain is not MSDND secure to either a cyber or physical level attack.*

This model is similar to Figure 6.7 and Figure 6.6 except that there is an information flow path from VO_2 to S_1 and CC_2 to CC_1 as seen in Figure 6.8.

Proof. Assume that in a cyber-physical blockchain, some function f_{φ_1} exists to determine whether φ_1 is true or false that is owned by CP and some function f_{φ_2} exists to determine whether φ_2 is true or false that is owned by CP. This follows from Assumption 7. In the model, it is assumed that CP will always be honest due to the bounded number of attacker in Assumption 3. Thus, since CP is the only entity with an information flow to LB_2 and LB_2 is the only entity with information flow to FP_2 it follows that if there exists a world such

that $w \vDash V^{CP}_{\neg\varphi_1}(w) = true \implies w \vDash V^{FP_2}_{\neg\varphi_1}(w) = true$ and $w \vDash V^{CP}_{\neg\varphi_2}(w) = true \implies w \vDash V^{FP_2}_{\neg\varphi_2}(w) = true$. It follows from Theorem 4.7 and Theorem 4.8 that in a cyber-physical blockchain, neither cyber or physical actions of vehicle 2 can be successfully altered in order to deceive the future platoon of vehicle 2. □

Lemma 6.2.9.1. *A blockchain is only secure against an attack if it has a verification mechanism for attacks coming from that component in the system.*

This lemma shows the inherent weakness in many previous approaches to applying blockchains to cyber-physical systems.

6.3. COMPLEXITY ANALYSIS

In the previous section, it was shown that their solution met the real-time requirements of a VANET. In the previous approach, a vehicle's entire blockchain was transmitted so that it could be verified. Thus, it would eventually take too long and force the vehicles to re-certify with the CA. The proposed scheme outperforms prior work and also meets the same real-time requirements due to the fact it takes advantage of a group digital signature. This signature allows for all vehicles in the platoon to agree on a single block. Thus, only a single block is required to join the next platoon, saving considerable time. Thus, to demonstrate that this work meets the same requirements, an asymptotic complexity analysis comparing the two algorithms is performed. The Car Registration Protocol is excluded from the comparison since it does not have a real-time requirement; it can take as long as it takes for the car to be physically inspected. The Intra-Platoon Communication protocol is also excluded due to the similarities between both approaches.

For this analysis, assume that hash calculation, generation of a random number, and basic math operations such as multiplication, division, addition, and subtraction take constant time. Let E be the time to perform exponentiation. Let M be the amount of data in a message divided by the bandwidth of the channel, S be the time for a vehicle to check

it's sensors, B be the time to send an entire block of a vehicle's blockchain, K be the time to verify that a block is correct and calculate a trust score for a block, H be the number of blocks in the entire history of a vehicle, and T be the time to create a new transaction.

6.3.1. Car Registration Protocol. Although a specific digital signature scheme method was not mentioned, we will assume that the scheme used ECDSA. The Car Registration Protocol is not bounded by any real-time requirements since no physical maneuvers happen during this protocol.

$$C_{Registration_{NEW}} = Gen_{Schnorr} + Gen_{ECDSA} + \Theta M + Verify + Create_{Block} + Log$$

$$C_{Registration_{NEW}} = \Theta(1) + E$$

6.3.2. Join Platoon Protocol. In the past approach, the entire blockchain of a vehicle was broadcast to the platoon it is requesting to join to be verified. Thus, the limiting factor of the old approach was that every vehicle must get recertified with the CA to receive a new blockchain once they receive enough block to exceed the real-time threshold. The proposed approach does not require the transmission of the entire blockchain. Instead, a joining vehicle simply needs to send its secure truncated block to the joining platoon.

Previous Method: $C_{Join} = 2M + HB + HK + S = \Theta(H(B + K)) + \Theta(1)$

Proposed Method: $C_{Join} = 2M + B + K + S = \Theta(1)$

Since all of the variables in the asymptotic complexity for the proposed method are constant, the asymptotic complexity is $\Theta(1)$. This faster than the previous message which is bounded asymptotically by the length of the history of a vehicle.

6.3.3. Block Creation Protocol. In the previous approach, a platoon leader selected a member of the platoon to use the PoW consensus mechanism so generate a block for the platoon. The purpose was the save the resources of the majority of the platoon while not giving too much control over the platoon blockchain to the leader. In the proposed approach, all of the vehicles participate to generate a single group signature. This allows the entire platoon to verify the integrity of the block before it is published. In the proposed protocol, three different sets of messages are broadcast to the entire platoon. Every time this occurs, the approach from (Dwork *et al.*, 1988) is taken to reach consensus. Algorithm 2 from (Dwork *et al.*, 1988) is used since the protocol can assume processors with bounded drift due to the secure GPS units located in each vehicle. Thus, consensus on each of these messages can be achieved in $3(4N\Phi + \Delta + 4\Phi)$ where N is the number of vehicles in the platoon, Φ is the upper bound on relative processor speed, and Δ is the upper bound on message delivery time which is equal to M.

Previous Method: $C_{Block} = 3M + 2\Theta(1)$

Proposed Method: $C_{Block} = M + E(N+1) + \Theta(1) + 9(4N\Phi + \Delta + 4\Phi) = 10M + E(N+1) + \Theta(1)$

Since Φ is negligible, the cost of the proposed method can be reduced. Thus, the cost of the proposed protocol slower than the previous approach.

6.3.4. Platoon Leave Protocol. In the proposed work, a vehicle must request to leave the platoon and wait for the platoon to create a new block for the vehicle before it is allowed to leave. Previously, the vehicle only needed to request to leave and wait for an acknowledgment before beginning the process.

Previous Method: $C_{Leave} = 2M + S$

Proposed Method: $C_{Leave} = 2M + S + (10M + E(N + 1) + \Theta(1))$

Thus, the proposed method is increased by the amount of time it takes to create a block. Since the M values are negligible as seen in the previous section, the cost of the protocol is dominated by the value for S. Thus, the proposed protocol is slowed by a cost of $N + 1$ exponentiation operations. Since the size of the platoons is bounded, the costs of both approaches are comparable.

7. CONCLUSION AND FUTURE DIRECTIONS

This dissertation presented a secure blockchain authentication scheme for VANETs that uses private blockchains representing the history of a vehicle and is used as a token to join future platoons. An efficient version of the algorithm that uses the Schnorr digital signature scheme to create a group signature that is signed by the entire platoon is also presented. This scheme was proven to be secure under a bounded number of attackers. The consensus mechanism presented uses basic Byzantine fault-tolerant algorithms to reach an agreement by the platoon during the block creation algorithm. This scheme provides significant cost savings over other solutions by reducing infrastructure components and removing all requirements of RSUs. Additionally, it was shown to meet real-time speed requirements via a complexity analysis comparison between the proposed protocols and previous protocols that were proven to meet real-time requirements.

This dissertation also showed the need for verification mechanisms when applying blockchains to cyber-physical systems. Without these, some level of trust must be assumed between users, contradicting a primary goal of blockchains. The verification mechanism presented in this dissertation was the use of other vehicles' sensors within a platoon to determine if a vehicle's actions are correct or incorrect. This approach leveraged the redundancy of sensor readings within a platoon to compare similar readings to determine vehicle behavior.

The authentication scheme solves the five problems outlines in Section 3. This dissertation allows the storage of physical-level data through the use of redundant data sources. Multiple sources of data, each with the knowledge of the ground truth, are then compared with one-another and majority consensus is reached. This allows for the verification of all information stored within the blockchain, maintaining a key benefit of blockchain technology.

The proposed solution solves the speed issues related to a system-wide ledger and the consensus mechanism by changing the general architecture of the blockchain. Instead of reaching consensus on a global level, local consensus on the platoon level is reached. Blocks are created, verified, and stored locally and used as a proof of behavior to join the next platoon. The added benefit of this approach is the cost savings caused by the removal of RSUs from the VANET architecture. A new consensus mechanism is introduced that leverages Schnorr digital signatures to remove the inherent risk of the 51% attack and further address real-time requirements by reaching consensus in a deterministic time compared to the probabilistic approach taken by past solutions.

Lastly, a CA is added to the system to handle initial registration for participants and extend an initial amount of trust to a vehicle. This removes the ability of an adversary to perform a Sybil attack while addressing the basic safety regulations in place for many cyber-physical systems.

Future research topics should include alternate verification methods for real-world data in blockchains, additional applications where blockchains can be leveraged to secure real-world data, future generalized approaches that allow a standard method for securing real-world data in blockchains, and performing experiments of this proposed approach on a real-time VANET system.

One potential alternate verification mechanism for real-world data is applying physical-level invariants within a blockchain to verify data. A significant amount of work has been done to apply physical-level invariants to increase the security in cyber-physical systems (Roth, 2015). These physical-level invariants could potentially be coded into a blockchain and forcibly used via smart contracts to automatically verify transactions. Significant research in the ability to meet real-time criteria must accompany any new approaches.

Applications of blockchains to secure cyber-physical invariants could be expanded to any applications where different entities must come together to complete some tasks. This includes transactive energy management, VANETS, supply-chain management, and industrial processes that could potentially be applications that could leverage from blockchains.

The work presented in this dissertation solved how to secure real-world data for one application: VANETs. However, this proposed approach may not apply to all cyber-physical systems. Most systems do not have redundancy in sensor reading that is inherent to the system. Thus, future work should include work towards a generalized approach for securing this data. The required traits and properties for these approaches should also be detailed.

This dissertation proved that the proposed verification mechanism met real-time requirements with complexity analysis and sample mathematical calculations. However, this approach and any future approaches like it need to be implemented on a real system and be evaluated to ensure that they meet real-time requirements. This is important for any cyber-physical system to prove that the methods do not result in loss of property, money, or life.

APPENDIX A.

SPIN MODEL CHECKER CODE

The Promela code used to verify the three stages of the a vehicle and their relationship with a platoon: Joining, Intra-Communication, and Leaving.

1. PLATOON JOIN PROMELA CODE

```
/*Define a vehicle data structure.*/
typedef vehStruct {
        chan mChannel = [30] of {mtype, byte, bool};
        byte id;
        bool posRecert;
        bool valVehFlag;
        bool inPlatFlag;
        short platDist;
        bool GPSOutput;
        bool brJoin;
        bool platLeader;
        bool restrictedStatus;
};

mtype = {Accept, Reject, RequestJoin, RequestBlock, BlockChain, BlockChainVerify, GPSRequest, GPSReply,
    InRange, AcceptRestricted};

/*-------------------------------------------------------------------------------------------------------------------------------------------*/
                        /*These are "global" variables.*/
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
byte platoonScore [10];
byte platoonCreated = 0;

bool validCommand = true;
bool blockCreationEvent = false;

byte doneVehicles = 0;
byte numVehicles = 0;
byte totVehicles = 5;
byte joinVehicles = 4;
byte numPlatoon = 0;                    /*This is the total of number of vehicles that should be present in this run.*/
vehStruct allVehicles[6];
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
                        /*Set up the initial state of the simulation*/
/*-------------------------------------------------------------------------------------------------------------------------------------------*/

init
{
atomic{
/* This is the list of initial system states for every vehicle*/
allVehicles[0].id = 0;
allVehicles[0].posRecert = true;
allVehicles[0].valVehFlag = true;
allVehicles[0].platDist = 0;
allVehicles[0].GPSOutput = true;
allVehicles[0].platLeader = true;
```

```
allVehicles[0].inPlatFlag = true;
allVehicles[0].brJoin = true;
allVehicles[0].restrictedStatus = false;
numPlatoon++;
printf("Vehicle %d is in the platoon\n",0);
numVehicles++;

allVehicles[1].id = 1;
allVehicles[1].posRecert = true;
allVehicles[1].valVehFlag = true;
allVehicles[1].platDist = 0;
allVehicles[1].GPSOutput = true;
allVehicles[1].platLeader = false;
allVehicles[1].inPlatFlag = false;
allVehicles[1].brJoin = false;
allVehicles[1].restrictedStatus = false;
printf("Vehicle %d is NOT in the platoon\n",1);
numVehicles++;

allVehicles[2].id = 2;
allVehicles[2].posRecert = true;
allVehicles[2].valVehFlag = true;
allVehicles[2].platDist = 0;
allVehicles[2].GPSOutput = true;
allVehicles[2].platLeader = false;
allVehicles[2].inPlatFlag = false;
allVehicles[2].brJoin = false;
allVehicles[2].restrictedStatus = false;
printf("Vehicle %d is NOT in the platoon\n",2);
numVehicles++;

allVehicles[3].id = 3;
allVehicles[3].posRecert = true;
allVehicles[3].valVehFlag = true;
allVehicles[3].platDist = 0;
allVehicles[3].GPSOutput = true;
allVehicles[3].platLeader = false;
allVehicles[3].inPlatFlag = false;
allVehicles[3].brJoin = false;
allVehicles[3].restrictedStatus = false;
printf("Vehicle %d is NOT in the platoon\n",3);
numVehicles++;

allVehicles[4].id = 4;
allVehicles[4].posRecert = false;
allVehicles[4].valVehFlag = true;
allVehicles[4].platDist = 0;
allVehicles[4].GPSOutput = true;
allVehicles[4].platLeader = false;
allVehicles[4].inPlatFlag = false;
allVehicles[4].brJoin = false;
allVehicles[4].restrictedStatus = false;
printf("Vehicle %d is NOT in the platoon\n",4);
numVehicles++;
}
```

```
run platVehicle(0);

byte counter = 1;

do
:: counter <= joinVehicles -> run vehicle(counter);  counter = counter + 1;
od

}

/*-----------------------------------------------------------------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------------------------------------------------------------*/
                        /*Broadcast Message Protocol*/
/*-----------------------------------------------------------------------------------------------------------------------------------*/
inline broadcast(message,id,flag)
{
        printf("Sendings %e, %d, %d\n",message,id,flag);
        byte temp_chan = 0;
        do
        :: (temp_chan < numVehicles) ->
                if
                :: (temp_chan!=input_id) ->
                        if
                        :: ((message == RequestBlock || message == Reject || message == Accept || message == GPSRequest)
                            && allVehicles[temp_chan].inPlatFlag == false) ->
                                //printf("temp_chan: %d, message: %e, id: %d, flag: %d\n",temp_chan,message,id,flag);
                                allVehicles[temp_chan].mChannel!message,id,flag;
                                printf("V%d sent %e,V%d to V%d\n",input_id,message,id,temp_chan);
                                //printf("L%d\n",len(allVehicles[temp_chan].mChannel));
                                temp_chan++;
                        :: ((message == RequestJoin || message == BlockChain || message == GPSReply)
                            && allVehicles[temp_chan].platLeader == true) ->
                                //printf("temp_chan: %d, message: %e, id: %d, flag: %d\n",temp_chan,message,id,flag);
                                allVehicles[temp_chan].mChannel!message,id,flag;
                                printf("V%d sent %e,V%d to V%d\n",input_id,message,id,temp_chan);
                                //printf("L%d\n",len(allVehicles[temp_chan].mChannel));
                                temp_chan++;
                        :: else -> temp_chan++;
                        fi
                :: (temp_chan==input_id) -> temp_chan++;
                fi
        :: else -> break;
        od;

}
/*-----------------------------------------------------------------------------------------------------------------------------------*/
                        /*Join Platoon Protocol*/
/*-----------------------------------------------------------------------------------------------------------------------------------*/
inline joinPlatoon()
{
        byte temp_id;

        /*Broadcast a request to join the platoon.*/
        atomic{broadcast(RequestJoin,input_id,true);}
        allVehicles[input_id].brJoin = true;
```

```
/*Wait until you receive a Request Block or a Reject Message*/
do
:: allVehicles[input_id].mChannel?<msg,id,flag> ->
        if
        :: (id == input_id && msg == RequestBlock) ->
                atomic{allVehicles[input_id].mChannel?msg,id,flag
                printf("msg is RequestBlock, id is %d, and flag is %d\n",input_id,flag);
                break;}
        /*:: (id == input_id && msg == Reject) ->
                atomic{allVehicles[input_id].mChannel?msg,id,flag;
                goto done}*/
        :: (id != input_id) ->
                atomic{printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                allVehicles[input_id].mChannel?msg,id,flag}
        :: else ->
                atomic{
                if
                :: allVehicles[input_id].mChannel?msg,id,flag ->
                        printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                :: len(allVehicles[input_id].mChannel)==0
                    -> printf("Error, V%d, msg is %e id is %d\n",input_id,msg,id)
                fi
                }
        fi
:: (len(allVehicles[input_id].mChannel) == 0) -> printf("V%d Waiting RB\n",input_id)
od

/*Broadcast your blockchain as well as your "recertification*/
atomic{broadcast(BlockChain,input_id,allVehicles[input_id].posRecert);}

/*Wait until you receive an Accept or GPS Message*/
do
:: allVehicles[input_id].mChannel?<msg,id,flag> ->
        if
        :: (id == input_id && msg == GPSRequest) ->
                atomic{
                allVehicles[input_id].mChannel?msg,id,flag
                printf("msg is GPSRequest, id is %d, and flag is %d\n",input_id,flag);
                break;
                }
        /*:: (id == input_id && msg == Reject) ->
                atomic{
                allVehicles[input_id].mChannel?msg,id,flag;
                goto done
                }*/
        :: (id != input_id) ->
                atomic{
                printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                allVehicles[input_id].mChannel?msg,id,flag
                }
        :: else ->
                atomic{
                if
                :: allVehicles[input_id].mChannel?msg,id,flag ->
                        printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                :: len(allVehicles[input_id].mChannel)==0
```

```
                        ->printf("Error, V%d, msg is %e id is %d\n",input_id,msg,id)
                    fi
                    }
            fi
        :: len(allVehicles[input_id].mChannel)==0 -> printf("V%d Waiting GPSR\n",input_id)
        od

        /*Broadcast your GPS output to the platoon*/
        atomic{broadcast(GPSReply,input_id,allVehicles[input_id].GPSOutput);}

        /*Wait until you are accepted or AcceptRestricted from the platoon*/
        do
        :: allVehicles[input_id].mChannel?<msg,id,flag> ->
            if
            :: (id == input_id && msg == Accept) ->
                    atomic{allVehicles[input_id].mChannel?msg,id,flag
                    printf("msg is Accept, id is %d, and flag is %d\n",input_id,flag);
                    break;}
            :: (id == input_id && msg == AcceptRestricted) ->
                    atomic{allVehicles[input_id].mChannel?msg,id,flag
                    printf("msg is Accept, id is %d, and flag is %d\n",input_id,flag);
                    allVehicles[input_id].restrictedStatus = true;
                    break;}
            /*:: (id == input_id && msg == Reject) ->
                    atomic{allVehicles[input_id].mChannel?msg,id,flag;
                    goto done}*/
            :: (id != input_id) ->
                    atomic{printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                    allVehicles[input_id].mChannel?msg,id,flag}
            :: else ->
                    atomic{
                    if
                    :: allVehicles[input_id].mChannel?msg,id,flag ->
                            printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                    :: len(allVehicles[input_id].mChannel)==0
                        -> printf("Error, V%d, msg is %e id is %d\n",input_id,msg,id)
                    fi}
            fi
        :: len(allVehicles[input_id].mChannel)==0 -> printf("V%d Waiting ACCEPT\n",input_id)
        od

        /*Set this vehicles inPlatFlag to true and print a statment.*/
        atomic{
        allVehicles[input_id].inPlatFlag = true;
        numPlatoon++;
        printf("Vehicle %d has join the platoon.\n",id)
        }
}
/*-------------------------------------------------------------------------------------------------------------------------*/
                    /*In-Platoon Car Protocol*/
/*-------------------------------------------------------------------------------------------------------------------------*/
proctype platVehicle(int input_id)
{
        byte id;
        bool flag;
        mtype msg;
```

```
do
:: allVehicles[input_id].mChannel?<msg,id,flag> ->
        if
        :: (msg == RequestJoin) ->
                atomic{
                allVehicles[input_id].mChannel?RequestJoin,id,flag;
                printf("V%d, msg is RequestJoin, id is %d, and flag is %d\n",input_id,id,flag)
                if
                :: (allVehicles[id].platDist == 0 && allVehicles[input_id].inPlatFlag == true)
                    -> atomic{broadcast(RequestBlock,id,true);}
                :: (allVehicles[id].platDist != 0 && allVehicles[input_id].inPlatFlag == true)
                    -> atomic{broadcast(Reject,id,true);}
                :: else
                fi
                }
        /*Check if you have a blockchain message in your channel*/
        :: (msg == BlockChain) ->
                atomic{
                allVehicles[input_id].mChannel?BlockChain,id,flag
                printf("msg is BlockChain, id is %d, and flag is %d, %d, %d\n",id,flag,allVehicles[id].platDist,
                    allVehicles[input_id].inPlatFlag)
                if
                :: (allVehicles[input_id].inPlatFlag == true) -> atomic{broadcast(GPSRequest,id,true);}
                //:: (flag!=true && allVehicles[input_id].inPlatFlag == true) ->
                    atomic{broadcast(Reject,id,true);}
                :: else
                fi
                }
        /*Check if you have a GPSReply in your channel*/
        :: (msg == GPSReply) ->
                atomic{
                allVehicles[input_id].mChannel?GPSReply,id,flag;
                printf("V%d, msg is GPSReply, id is %d, and flag is %d\n",input_id,id,flag);
                if
                :: (flag==true && allVehicles[id].GPSOutput==true && allVehicles[input_id].inPlatFlag == true) ->
                 atomic{broadcast(Accept,id,true);}
                :: ((flag!=true || allVehicles[id].GPSOutput!=true) && allVehicles[input_id].inPlatFlag == true) ->
                    atomic{broadcast(AcceptRestricted,id,true);}
                :: else
                fi
                }
        :: else ->
                atomic{
                allVehicles[input_id].mChannel?msg,id,flag;
                printf("Message Deleted: V:%d M:%e, ID:%d\n",input_id,msg,id);
                }
        fi
/*If the number of live processes is equal to the number of vehicle in the platoon then exit*/
:: (numVehicles == numPlatoon) -> break;
//:: (len(allVehicles[input_id].mChannel) == 0 && !(numVehicles == numPlatoon))->
//      printf("Empty Channel for V%d. NumPlat: %d, numVehicles: %d, veh1flag: %d, veh2flag: %d, veh3flag: %d,
    veh4flag:%d\n",input_id,numPlatoon,numVehicles,allVehicles[0].inPlatFlag,allVehicles[1].inPlatFlag,
    allVehicles[2].inPlatFlag,allVehicles[3].inPlatFlag);
od
```

```
doneVehicles++;
done:
        printf("Vehicle %d is done.\n",input_id);
}

/*-----------------------------------------------------------------------------------------------------------------------------------------------------------*/
                        /*Car Protocol*/
/*-----------------------------------------------------------------------------------------------------------------------------------------------------------*/
proctype vehicle(int input_id)
{
        byte id;
        bool flag;
        mtype msg;

        do
        /*Check if the vehicle is in the platoon. If not then try to join*/
        :: (allVehicles[input_id].inPlatFlag == false) -> joinPlatoon(); allVehicles[input_id].brJoin = true;
        /*Need to add step where platoon vehicles communicate if they see a vehicle*/
        /*Check if you have a RequestJoin message in the message channel*/
        :: allVehicles[input_id].mChannel?<msg,id,flag> ->
                if
                :: (msg == RequestJoin) ->
                        atomic{
                        allVehicles[input_id].mChannel?RequestJoin,id,flag;
                        printf("V%d, msg is RequestJoin, id is %d, and flag is %d\n",input_id,id,flag)
                        if
                        :: (allVehicles[id].platDist == 0 && allVehicles[input_id].inPlatFlag == true)
                            -> atomic{broadcast(RequestBlock,id,true);}
                        :: (allVehicles[id].platDist != 0 && allVehicles[input_id].inPlatFlag == true)
                            -> atomic{broadcast(Reject,id,true);}
                        :: else
                        fi}
                /*Check if you have a blockchain message in your channel*/
                :: (msg == BlockChain) ->
                        atomic{
                        allVehicles[input_id].mChannel?BlockChain,id,flag
                        printf("msg is BlockChain, id is %d, and flag is %d, %d, %d\n",id,flag,allVehicles[id].platDist,
                            allVehicles[input_id].inPlatFlag)
                        if
                        :: (allVehicles[input_id].inPlatFlag == true) -> atomic{broadcast(GPSRequest,id,true);}
                        //:: (flag!=true && allVehicles[input_id].inPlatFlag == true) -> atomic{broadcast(Reject,id,true);}
                        :: else
                        fi}
                /*Check if you have a GPSReply in your channel*/
                :: (msg == GPSReply) ->
                        atomic{
                        allVehicles[input_id].mChannel?GPSReply,id,flag
                        printf("V%d, msg is GPSReply, id is %d, and flag is %d\n",input_id,id,flag)
                        if
                        :: (flag==true && allVehicles[input_id].inPlatFlag == true) -> atomic{broadcast(Accept,id,true);}
                        :: (flag!=true && allVehicles[input_id].inPlatFlag == true) -> atomic{broadcast(AcceptRestricted,id,true);}
                        :: else
                        fi
                        }
                :: else ->
```

```
                        atomic{
                        allVehicles[input_id].mChannel?msg,id,flag;
                        printf("Message Deleted: V:%d M:%e, ID:%d\n",input_id,msg,id);
                        }
                fi
        /*If the number of live processes is equal to the number of vehicle in the platoon then exit*/
        :: (numVehicles == numPlatoon) -> break;
        :: (len(allVehicles[input_id].mChannel) == 0 && !(numVehicles == numPlatoon) && !(allVehicles[input_id].inPlatFlag == false))->
                printf("Empty Channel for V%d. NumPlat: %d, numVehicles: %d, veh1flag: %d, veh2flag: %d, veh3flag: %d,
                    veh4flag:%d\n",input_id,numPlatoon,numVehicles,allVehicles[0].inPlatFlag,allVehicles[1].inPlatFlag,
                    allVehicles[2].inPlatFlag,allVehicles[3].inPlatFlag);
        od

doneVehicles++;
do
:: (totVehicles == doneVehicles) -> break;
od

done:
        printf("Vehicle %d is done.\n",input_id);

}

/*-----------------------------------------------------------------------------------------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------------------------------------------------------------------------------------*/
                        /*These are the verification variables.*/
/*-----------------------------------------------------------------------------------------------------------------------------------------------------------*/
#define phy1 (allVehicles[0].platDist == 0)
#define cp1 (allVehicles[0].posRecert == true)
#define br1 (allVehicles[0].brJoin == true)
#define gps1 (allVehicles[0].GPSOutput == true)
#define pj1 (allVehicles[0].inPlatFlag == true)
#define rs1 (allVehicles[0].restrictedStatus==true)

#define phy2 (allVehicles[1].platDist == 0)
#define cp2 (allVehicles[1].posRecert == true)
#define br2 (allVehicles[1].brJoin == true)
#define gps2 (allVehicles[1].GPSOutput == true)
#define pj2 (allVehicles[1].inPlatFlag == true)
#define rs2 (allVehicles[1].restrictedStatus==true)

#define phy3 (allVehicles[2].platDist == 0)
#define cp3 (allVehicles[2].posRecert == true)
#define br3 (allVehicles[2].brJoin == true)
#define gps3 (allVehicles[2].GPSOutput == true)
#define pj3 (allVehicles[2].inPlatFlag == true)
#define rs3 (allVehicles[2].restrictedStatus==true)

#define phy4 (allVehicles[3].platDist == 0)
#define cp4 (allVehicles[3].posRecert == true)
#define br4 (allVehicles[3].brJoin == true)
#define gps4 (allVehicles[3].GPSOutput == true)
#define pj4 (allVehicles[3].inPlatFlag == true)
#define rs4 (allVehicles[3].restrictedStatus==true)

#define phy5 (allVehicles[4].platDist == 0)
```

```
#define cp5 (allVehicles[4].posRecert == true)
#define br5 (allVehicles[4].brJoin == true)
#define gps5 (allVehicles[4].GPSOutput == true)
#define pj5 (allVehicles[4].inPlatFlag == true)
#define rs5 (allVehicles[4].restrictedStatus==true)
/*-----------------------------------------------------------------------------------------------------------------------------------------*/
```

2. PLATOON INTRA-COMMUNICATION PROMELA CODE

```
/*Define a vehicle data structure.*/
typedef vehStruct {
        chan mChannel = [30] of {mtype, byte, bool};     /*A Channel which holds 16 messages of which consist of two byte fields.*/
        byte id;
    byte transactionsReceived = 0;
        bool posRecert;
        bool valVehFlag;
        bool inPlatFlag;
        short platDist;
        bool GPSOutput;
        bool brJoin;
        bool platLeader;
    bool followCommand;
        bool restrictedStatus;
        bool newBlock;
};

mtype = {Accept, Reject, Command, Evaluate, Transaction, CreateBlock, Block };

/*-----------------------------------------------------------------------------------------------------------------------------------------*/
                        /*These are "global" variables.*/
/*-----------------------------------------------------------------------------------------------------------------------------------------*/
byte platoonScore [10];
byte platoonCreated = 0;

bool validCommand = true;
bool blockCreationEvent = false;

byte receivedBlock = 0;
byte performedCommand = 0;
byte doneVehicles = 0;
byte numVehicles = 0;
byte totVehicles = 5;
byte joinVehicles = 4;
byte numPlatoon = 0;                    /*This is the total of number of vehicles that should be present in this run.*/
vehStruct allVehicles[6];
/*-----------------------------------------------------------------------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------------------------------------------------------------------*/
                        /*Set up the initial state of the simulation*/
/*-----------------------------------------------------------------------------------------------------------------------------------------*/

init
{
atomic{
/* This is the list of initial system states for every vehicle*/
```

```
allVehicles[0].id = 0;
allVehicles[0].posRecert = true;
allVehicles[0].valVehFlag = true;
allVehicles[0].platDist = 0;
allVehicles[0].GPSOutput = true;
allVehicles[0].platLeader = true;
allVehicles[0].inPlatFlag = true;
allVehicles[0].brJoin = true;
allVehicles[0].followCommand = true;
allVehicles[0].restrictedStatus = false;
numPlatoon++;
printf("Vehicle %d is in the platoon\n",0);
numVehicles++;

allVehicles[1].id = 1;
allVehicles[1].posRecert = true;
allVehicles[1].valVehFlag = true;
allVehicles[1].platDist = 0;
allVehicles[1].GPSOutput = true;
allVehicles[1].platLeader = false;
allVehicles[1].inPlatFlag = true;
allVehicles[1].brJoin = true;
allVehicles[1].followCommand = true;
allVehicles[1].restrictedStatus = false;
printf("Vehicle %d is in the platoon\n",1);
numVehicles++;
numPlatoon++;

allVehicles[2].id = 2;
allVehicles[2].posRecert = true;
allVehicles[2].valVehFlag = true;
allVehicles[2].platDist = 0;
allVehicles[2].GPSOutput = true;
allVehicles[2].platLeader = false;
allVehicles[2].inPlatFlag = true;
allVehicles[2].brJoin = true;
allVehicles[2].followCommand = true;
allVehicles[2].restrictedStatus = false;
printf("Vehicle %d is in the platoon\n",2);
numVehicles++;
numPlatoon++;

allVehicles[3].id = 3;
allVehicles[3].posRecert = true;
allVehicles[3].valVehFlag = true;
allVehicles[3].platDist = 0;
allVehicles[3].GPSOutput = true;
allVehicles[3].platLeader = false;
allVehicles[3].inPlatFlag = true;
allVehicles[3].brJoin = true;
allVehicles[3].followCommand = false;
allVehicles[3].restrictedStatus = false;
printf("Vehicle %d is in the platoon\n",3);
numVehicles++;
numPlatoon++;
```

```
allVehicles[4].id = 4;
allVehicles[4].posRecert = true;
allVehicles[4].valVehFlag = true;
allVehicles[4].platDist = 0;
allVehicles[4].GPSOutput = true;
allVehicles[4].platLeader = false;
allVehicles[4].inPlatFlag = true;
allVehicles[4].brJoin = true;
allVehicles[4].followCommand = true;
allVehicles[4].restrictedStatus = false;
printf("Vehicle %d is in the platoon\n",4);
numVehicles++;
numPlatoon++;
}
run platVehicle(0);

byte counter = 1;

do
:: counter <= joinVehicles -> run vehicle(counter);  counter = counter + 1;
od

}

/*-----------------------------------------------------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------------------------------------------------*/
                        /*Broadcast Message Protocol*/
/*-----------------------------------------------------------------------------------------------------------------------*/
inline broadcast(message,id,flag)
{
        printf("Sendings %e, %d, %d\n",message,id,flag);
        byte temp_chan = 0;
        do
        :: (temp_chan < numVehicles) ->
                if
                :: (temp_chan!=input_id) ->
                        if
                        :: ((message == Command || message == Evaluate || message == CreateBlock || message ==
                        Transaction || message == Block) && allVehicles[temp_chan].platLeader == false) ->
                                //printf("temp_chan: %d, message: %e, id: %d, flag: %d\n",temp_chan,message,id,flag);
                                allVehicles[temp_chan].mChannel!message,id,flag;
                                printf("V%d sent %e,V%d to V%d\n",input_id,message,id,temp_chan);
                                //printf("L%d\n",len(allVehicles[temp_chan].mChannel));
                                temp_chan++;
                        :: ((message == Reject || message == Transaction || message == Block)
                            && allVehicles[temp_chan].platLeader == true) ->
                                //printf("temp_chan: %d, message: %e, id: %d, flag: %d\n",temp_chan,message,id,flag);
                                allVehicles[temp_chan].mChannel!message,id,flag;
                                printf("V%d sent %e,V%d to V%d\n",input_id,message,id,temp_chan);
                                //printf("L%d\n",len(allVehicles[temp_chan].mChannel));
                                temp_chan++;
                        :: else -> temp_chan++;
                        fi
                :: (temp_chan==input_id) -> temp_chan++;
                fi
        :: else -> break;
```

```
        od;

}
/*-------------------------------------------------------------------------------------------------------------------------------------*/
                        /*Evaluate Message Protocol*/
/*-------------------------------------------------------------------------------------------------------------------------------------*/
inline evaluatePlatoon()
{
    byte temp_id_2 = 0;
    do
    :: (temp_id_2 < numVehicles && allVehicles[temp_id_2].followCommand == true) ->
        platoonScore[temp_id_2] = platoonScore[temp_id_2] + 1; temp_id_2 = temp_id_2 + 1;
    :: (temp_id_2 < numVehicles && allVehicles[temp_id_2].followCommand == false) ->
        platoonScore[temp_id_2] = platoonScore[temp_id_2] - 1; temp_id_2 = temp_id_2 + 1;
    :: else -> break;
    od;

    atomic{broadcast(Transaction, input_id, true);};

}
/*-------------------------------------------------------------------------------------------------------------------------------------*/
                        /*Intra Platoon Protocol*/
/*-------------------------------------------------------------------------------------------------------------------------------------*/
inline intraPlatoon()
{
        byte temp_id;

        /*Broadcast a command to the platoon.*/
        atomic{broadcast(Command,input_id,true);
        allVehicles[input_id].followCommand=true;
        performedCommand++;}

        /*Wait until you receive a Reject Message or all vehicles follow the command.*/
        do
        :: allVehicles[input_id].mChannel?<msg,id,flag> ->
                if
                :: (id == input_id && msg == Reject) ->
                        atomic{allVehicles[input_id].mChannel?msg,id,flag;
                        goto done}
                :: (id != input_id) ->
                        atomic{printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                        allVehicles[input_id].mChannel?msg,id,flag}
                :: else ->
                        atomic{
                        if
                        :: allVehicles[input_id].mChannel?msg,id,flag ->
                                printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                        :: len(allVehicles[input_id].mChannel)==0
                            -> printf("Error, V%d, msg is %e id is %d\n",input_id,msg,id)
                        fi
                        }
                fi
        :: performedCommand == numVehicles -> break;
        :: (len(allVehicles[input_id].mChannel) == 0 && performedCommand != numVehicles)
            -> printf("V%d Waiting for platoon to perform the command.\n",input_id)
        od
```

```
    /*Broadcast an Evaluate command to the entire platoon.*/
    atomic{broadcast(Evaluate,input_id,allVehicles[input_id].posRecert);}

/*Evaluate the rest of the Platoon*/
evaluatePlatoon();
    allVehicles[input_id].transactionsReceived = allVehicles[input_id].transactionsReceived + 1;
    /*Wait until you receive an Tranasction Message*/
    do
    :: allVehicles[input_id].mChannel?<msg,id,flag> ->
            if
            :: (msg == Transaction) ->
                    atomic{
                    allVehicles[input_id].mChannel?msg,id,flag
                    printf("msg is Transaction, id is %d, and flag is %d\n",input_id,flag);
        allVehicles[input_id].transactionsReceived = allVehicles[input_id].transactionsReceived + 1;
                    }
            :: else ->
                    atomic{
                    if
                    :: allVehicles[input_id].mChannel?msg,id,flag ->
                            printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                    :: len(allVehicles[input_id].mChannel)==0
                        -> printf("Error, V%d, msg is %e id is %d\n",input_id,msg,id)
                    fi
                    }
            fi
:: allVehicles[input_id].transactionsReceived == numVehicles -> break;
    :: len(allVehicles[input_id].mChannel)==0 -> printf("V%d Waiting Transactions\n",input_id)
    od

    /*Broadcast a BlockCreation request to a platoon member*/
byte blockCreationId;

if
:: blockCreationId = 1;
:: blockCreationId = 2;
:: blockCreationId = 3;
:: blockCreationId = 4;
fi

    blockCreationEvent = true;
    atomic{broadcast(CreateBlock,blockCreationId,allVehicles[input_id].GPSOutput);}

    /*Wait until you recieve a block back*/
    do
    :: allVehicles[input_id].mChannel?<msg,id,flag> ->
            if
            :: (id == input_id && msg == Block && flag == true) ->
                    atomic{allVehicles[input_id].mChannel?msg,id,flag
                    printf("msg is a good Block, id is %d, and flag is %d\n",input_id,flag);
        receivedBlock = receivedBlock + 1;
                    break;}
    :: (id == input_id && msg == Block && flag == false) ->
                    if
        :: blockCreationId = 1;
```

```
            :: blockCreationId = 2;
            :: blockCreationId = 3;
            :: blockCreationId = 4;
            fi
                atomic{broadcast(CreateBlock,blockCreationId,allVehicles[input_id].GPSOutput);}
                        printf("msg is a bad Block, id is %d, and flag is %d\n",input_id,flag);
                :: (id != input_id) ->
                        atomic{printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                        allVehicles[input_id].mChannel?msg,id,flag}
                :: else ->
                        atomic{
                        if
                        :: allVehicles[input_id].mChannel?msg,id,flag ->
                                printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                        :: len(allVehicles[input_id].mChannel)==0
                            -> printf("Error, V%d, msg is %e id is %d\n",input_id,msg,id)
                        fi}
                fi
        :: len(allVehicles[input_id].mChannel)==0 -> printf("V%d Waiting Block\n",input_id)
        od

    /*Lastly, broadcast the new block to the platoon. */
    broadcast(Block, 10, true);
}
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
                        /*In-Platoon Car Protocol*/
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
proctype platVehicle(int input_id)
{

        byte id;
        bool flag;
        mtype msg;

    allVehicles[input_id].platLeader==true -> intraPlatoon();

doneVehicles++;
done:
        printf("Vehicle %d is done.\n",input_id);
}

/*-------------------------------------------------------------------------------------------------------------------------------------------*/
                        /*Car Protocol*/
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
proctype vehicle(int input_id)
{

        byte id;
        bool flag;
        mtype msg;

        do
        /*Check if you have a RequestJoin message in the message channel*/
        :: allVehicles[input_id].mChannel?<msg,id,flag> ->
                if
                 /*Check if the message is a command. If it is valid, perform it. Otherwise, reject it.*/
```

```
        :: (msg == Command && id == 0) ->
                atomic{
                allVehicles[input_id].mChannel?Command,id,flag;
                printf("V%d, msg is Command, id is %d, and flag is %d\n",input_id,id,flag)
                if
                :: (validCommand == true && allVehicles[input_id].inPlatFlag == true)
                    -> performedCommand++;
                :: (validCommand == false && allVehicles[input_id].inPlatFlag == true)
                    -> atomic{broadcast(Reject,id,false);}
                :: else
                fi
                }
        /*Check if you have a Evaluate message in your channel*/
        :: (msg == Evaluate && id == 0) ->
                atomic{
                allVehicles[input_id].mChannel?Evaluate,id,flag
                printf("msg is Evaluate, id is %d, and flag is %d, %d, %d\n",
                    id,flag,allVehicles[id].platDist,allVehicles[input_id].inPlatFlag)
                evaluatePlatoon();
                }
        /*Check if you have a Transaction in your channel*/
        :: (msg == Transaction && id == input_id) ->
                atomic{
                allVehicles[input_id].mChannel?msg,id,flag
                printf("msg is Transaction, id is %d, and flag is %d\n",input_id,flag);
    allVehicles[input_id].transactionsReceived = allVehicles[input_id].transactionsReceived + 1;
                }
        /*If it is a create block event, reassign the platoons recertifications*/
:: (msg == CreateBlock && id == input_id) ->
    atomic{
        allVehicles[input_id].mChannel?msg,id,flag


                /*go through all of the evaluations of the platoon and gather their scores.*/
                byte counter = 0;

                do
                :: platoonScore[counter] <= 0  && counter < numVehicles
                    -> allVehicles[counter].posRecert = false;
                    counter++;
                :: platoonScore[counter] > 0 && counter < numVehicles
                    -> allVehicles[counter].posRecert = true;
                    counter++;
                :: else -> break;
                od

                /*Broadcast the new block*/
                broadcast(Block,0,true);
                }
:: (msg == Block && id == 10) ->
    atomic{receivedBlock++;
        allVehicles[input_id].mChannel?msg,id,flag
    printf("New Block Received, V%d is waiting.\n",input_id,flag);
        break;}
:: else ->
                atomic{
```

```
                        allVehicles[input_id].mChannel?msg,id,flag;
                        printf("Message Deleted: V:%d M:%e, ID:%d\n",input_id,msg,id);
                        }
                fi
        :: (len(allVehicles[input_id].mChannel) == 0 && !(numVehicles == numPlatoon)
            && !(allVehicles[input_id].inPlatFlag == false))->
                printf("Empty Channel for V%d. NumPlat: %d, numVehicles: %d, veh1flag: %d, veh2flag: %d, veh3flag: %d,
                    veh4flag:%d\n",input_id,numPlatoon,numVehicles,allVehicles[0].inPlatFlag,allVehicles[1].inPlatFlag,
                        allVehicles[2].inPlatFlag,allVehicles[3].inPlatFlag);
        od

doneVehicles++;
do
:: (totVehicles == doneVehicles) -> break;
od

done:
        printf("Vehicle %d is done.\n",input_id);

}

/*-----------------------------------------------------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------------------------------------------------*/
                        /*These are the verification variables.*/
/*-----------------------------------------------------------------------------------------------------------------------*/
#define bc (blockCreationEvent == true)

#define phy1 (allVehicles[0].platDist == 0)
#define cp1 (allVehicles[0].posRecert == true)
#define br1 (allVehicles[0].brJoin == true)
#define gps1 (allVehicles[0].GPSOutput == true)
#define pj1 (allVehicles[0].inPlatFlag == true)
#define bm1 (allVehicles[0].followCommand == false)
#define nb1 (allVehicles[0].newBlock == true)

#define phy2 (allVehicles[1].platDist == 0)
#define cp2 (allVehicles[1].posRecert == true)
#define br2 (allVehicles[1].brJoin == true)
#define gps2 (allVehicles[1].GPSOutput == true)
#define pj2 (allVehicles[1].inPlatFlag == true)
#define bm2 (allVehicles[1].followCommand == false)
#define nb2 (allVehicles[1].newBlock == true)

#define phy3 (allVehicles[2].platDist == 0)
#define cp3 (allVehicles[2].posRecert == true)
#define br3 (allVehicles[2].brJoin == true)
#define gps3 (allVehicles[2].GPSOutput == true)
#define pj3 (allVehicles[2].inPlatFlag == true)
#define bm3 (allVehicles[2].followCommand == false)
#define nb3 (allVehicles[2].newBlock == true)

#define phy4 (allVehicles[3].platDist == 0)
#define cp4 (allVehicles[3].posRecert == true)
#define br4 (allVehicles[3].brJoin == true)
#define gps4 (allVehicles[3].GPSOutput == true)
#define pj4 (allVehicles[3].inPlatFlag == true)
```

```
#define bm4 (allVehicles[3].followCommand == false)
#define nb4 (allVehicles[3].newBlock == true)

#define phy5 (allVehicles[4].platDist == 0)
#define cp5 (allVehicles[4].posRecert == true)
#define br5 (allVehicles[4].brJoin == true)
#define gps5 (allVehicles[4].GPSOutput == true)
#define pj5 (allVehicles[4].inPlatFlag == true)
#define bm5 (allVehicles[4].followCommand == false)
#define nb5 (allVehicles[4].newBlock == true)
/*-------------------------------------------------------------------------------------------------------------------------*/
```

3. PLATOON LEAVE PROEMLA CODE

```
/*Define a vehicle data structure.*/
typedef vehStruct {
        chan mChannel = [30] of {mtype, byte, bool};
        byte id;
        bool posRecert;
        bool valVehFlag;
        bool inPlatFlag;
        short platDist;
        bool GPSOutput;
        bool brJoin;
        bool platLeader;
        bool restrictedStatus
};

mtype = {Accept, Reject, RequestLeave, InRange};

/*-------------------------------------------------------------------------------------------------------------------------*/
                        /*These are "global" variables.*/
/*-------------------------------------------------------------------------------------------------------------------------*/
byte platoonScore [10];
byte platoonCreated = 0;

bool validCommand = true;
bool blockCreationEvent = false;

byte doneVehicles = 0;
byte numVehicles = 0;
byte totVehicles = 5;
byte leaveVehicles = 4;
byte numPlatoon = 0;            /*This is the total of number of vehicles that should be present in this run.*/
vehStruct allVehicles[6];
/*-------------------------------------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------------------------------------*/
                        /*Set up the initial state of the simulation*/
/*-------------------------------------------------------------------------------------------------------------------------*/

init
{
atomic{
/* This is the list of initial system states for every vehicle*/
```

```
allVehicles[0].id = 0;
allVehicles[0].posRecert = true;
allVehicles[0].valVehFlag = true;
allVehicles[0].platDist = 0;
allVehicles[0].GPSOutput = true;
allVehicles[0].platLeader = true;
allVehicles[0].inPlatFlag = true;
allVehicles[0].brJoin = true;
allVehicles[0].restrictedStatus = false;
printf("Vehicle %d is in the platoon\n",0);
numVehicles++;
numPlatoon++;

allVehicles[1].id = 1;
allVehicles[1].posRecert = true;
allVehicles[1].valVehFlag = true;
allVehicles[1].platDist = 0;
allVehicles[1].GPSOutput = true;
allVehicles[1].platLeader = false;
allVehicles[1].inPlatFlag = true;
allVehicles[1].brJoin = true;
allVehicles[1].restrictedStatus = false;
printf("Vehicle %d is in the platoon\n",1);
numVehicles++;
numPlatoon++;

allVehicles[2].id = 2;
allVehicles[2].posRecert = true;
allVehicles[2].valVehFlag = true;
allVehicles[2].platDist = 0;
allVehicles[2].GPSOutput = true;
allVehicles[2].platLeader = false;
allVehicles[2].inPlatFlag = true;
allVehicles[2].brJoin = true;
allVehicles[2].restrictedStatus = false;
printf("Vehicle %d is in the platoon\n",2);
numVehicles++;
numPlatoon++;

allVehicles[3].id = 3;
allVehicles[3].posRecert = true;
allVehicles[3].valVehFlag = true;
allVehicles[3].platDist = 0;
allVehicles[3].GPSOutput = true;
allVehicles[3].platLeader = false;
allVehicles[3].inPlatFlag = true;
allVehicles[3].brJoin = true;
allVehicles[3].restrictedStatus = false;
printf("Vehicle %d is in the platoon\n",3);
numVehicles++;
numPlatoon++;

allVehicles[4].id = 4;
allVehicles[4].posRecert = true;
allVehicles[4].valVehFlag = true;
allVehicles[4].platDist = 0;
```

```
allVehicles[4].GPSOutput = true;
allVehicles[4].platLeader = false;
allVehicles[4].inPlatFlag = true;
allVehicles[4].brJoin = true;
allVehicles[4].restrictedStatus = false;
printf("Vehicle %d is in the platoon\n",4);
numVehicles++;
numPlatoon++;
}

run platVehicle(0);

byte counter = 1;

do
:: counter <= leaveVehicles -> run vehicle(counter);  counter = counter + 1;
od

}

/*-------------------------------------------------------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
                        /*Broadcast Message Protocol*/
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
inline broadcast(message,id,flag)
{
        printf("Sendings %e, %d, %d\n",message,id,flag);
        byte temp_chan = 0;
        do
        :: (temp_chan < numVehicles) ->
                if
                :: (temp_chan!=input_id) ->
                        if
                        :: ((message == Reject || message == Accept) && allVehicles[temp_chan].inPlatFlag == true) ->
                                //printf("temp_chan: %d, message: %e, id: %d, flag: %d\n",temp_chan,message,id,flag);
                                allVehicles[temp_chan].mChannel!message,id,flag;
                                printf("V%d sent %e,V%d to V%d\n",input_id,message,id,temp_chan);
                                //printf("L%d\n",len(allVehicles[temp_chan].mChannel));
                                temp_chan++;
                        :: ((message == RequestLeave) && allVehicles[temp_chan].platLeader == true) ->
                                //printf("temp_chan: %d, message: %e, id: %d, flag: %d\n",temp_chan,message,id,flag);
                                allVehicles[temp_chan].mChannel!message,id,flag;
                                printf("V%d sent %e,V%d to V%d\n",input_id,message,id,temp_chan);
                                //printf("L%d\n",len(allVehicles[temp_chan].mChannel));
                                temp_chan++;
                        :: else -> temp_chan++;
                        fi
                :: (temp_chan==input_id) -> temp_chan++;
                fi
        :: else -> break;
        od;

}
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
                        /*Leave Platoon Protocol*/
/*-------------------------------------------------------------------------------------------------------------------------------------------*/
```

```
inline leavePlatoon()
{
        byte temp_id;

        /*Broadcast a request to leave the platoon.*/
        atomic{broadcast(RequestLeave,input_id,true);}

        /*Wait until you receive an Accept or a Reject Message*/
        do
        :: allVehicles[input_id].mChannel?<msg,id,flag> ->
                if
                :: (id == input_id && msg == Accept) ->
                        atomic{allVehicles[input_id].mChannel?msg,id,flag
                        printf("msg is Accept, id is %d, and flag is %d\n",input_id,flag);
                        break;}
                :: (id == input_id && msg == Reject) ->
                        atomic{allVehicles[input_id].mChannel?msg,id,flag;
                        goto done}
                :: (id != input_id) ->
                        atomic{printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                        allVehicles[input_id].mChannel?msg,id,flag}
                :: else ->
                        atomic{
                        if
                        :: allVehicles[input_id].mChannel?msg,id,flag ->
                                printf("V%d Removing, msg is %e id is %d\n",input_id,msg,id);
                        :: len(allVehicles[input_id].mChannel)==0
                            -> printf("Error, V%d, msg is %e id is %d\n",input_id,msg,id)
                        fi
                        }
                fi
        :: (len(allVehicles[input_id].mChannel) == 0) -> printf("V%d Waiting RB\n",input_id)
        od

        /*Set this vehicles inPlatFlag to false and print a statment.*/
        atomic{

        /*Set platDist to 1 noting that the vehicle has moved away from the platoon.*/
    allVehicles[input_id].platDist=1;

        allVehicles[input_id].inPlatFlag = false;
        numPlatoon--;
        printf("Vehicle %d has left the platoon.\n",id)
        }
}
/*-------------------------------------------------------------------------------------------------------------------------*/
                        /*In-Platoon Car Protocol*/
/*-------------------------------------------------------------------------------------------------------------------------*/
proctype platVehicle(int input_id)
{
        byte id;
        bool flag;
        mtype msg;


        do
```

```
        :: allVehicles[input_id].mChannel?<msg,id,flag> ->
                if
                :: (msg == RequestLeave) ->
                        atomic{
                        allVehicles[input_id].mChannel?RequestLeave,id,flag;
                        printf("V%d, msg is RequestLeave, id is %d, and flag is %d\n",input_id,id,flag)
                        if
                        :: (allVehicles[id].platDist == 0 && allVehicles[input_id].inPlatFlag == true)
                            -> atomic{broadcast(Accept,id,true);}
                        :: (allVehicles[id].platDist != 0 && allVehicles[input_id].inPlatFlag == true)
                            -> atomic{broadcast(Reject,id,true);}
                        :: else
                        fi
                        }
                :: else ->
                        atomic{
                        allVehicles[input_id].mChannel?msg,id,flag;
                        printf("Message Deleted: V:%d M:%e, ID:%d\n",input_id,msg,id);
                        }
                fi
        /*If the number of live processes is equal to the number of vehicle in the platoon then exit*/
        :: (numPlatoon == 1) -> break;
        od

doneVehicles++;
done:
        printf("Vehicle %d is done.\n",input_id);
}

/*-------------------------------------------------------------------------------------------------------------------------------------------------------*/
                        /*Car Protocol*/
/*-------------------------------------------------------------------------------------------------------------------------------------------------------*/
proctype vehicle(int input_id)
{
        byte id;
        bool flag;
        mtype msg;

    bool leaveFlag;
    if
    :: leaveFlag=true;
    /*:: leaveFlag=false;*/
    fi

        do
        /*Check if the vehicle is in the platoon. If so then try to leave*/
        :: (leaveFlag == true) -> leavePlatoon(); break;
        /*Need to add step where platoon vehicles communicate if they see a vehicle*/
        /*Check if you have a RequestLeave message in the message channel*/
        :: allVehicles[input_id].mChannel?<msg,id,flag> ->
                if
                :: (msg == RequestLeave) ->
                        atomic{
                        allVehicles[input_id].mChannel?RequestLeave,id,flag;
                        printf("V%d, msg is RequestLeave id is %d, and flag is %d\n",input_id,id,flag)
                        if
```

```
                        :: (allVehicles[id].platDist == 0 && allVehicles[input_id].inPlatFlag == true)
                            -> atomic{broadcast(Accept,id,true);}
                        :: (allVehicles[id].platDist != 0 && allVehicles[input_id].inPlatFlag == true)
                            -> atomic{broadcast(Reject,id,true);}
                        :: else
                        fi}
                :: else ->
                        atomic{
                        allVehicles[input_id].mChannel?msg,id,flag;
                        printf("Message Deleted: V:%d M:%e, ID:%d\n",input_id,msg,id);
                        }
                fi
        /*If the number of live processes is equal to the number of vehicle in the platoon then exit*/
        /*:: (numVehicles == numPlatoon) -> break;*/
        :: (len(allVehicles[input_id].mChannel) == 0
            && !(numVehicles == numPlatoon) && !(allVehicles[input_id].inPlatFlag == false))->
                printf("Empty Channel for V%d. NumPlat: %d, numVehicles: %d, veh1flag: %d, veh2flag: %d, veh3flag: %d,
                    veh4flag:%d\n",input_id,numPlatoon,numVehicles,allVehicles[0].inPlatFlag,allVehicles[1].inPlatFlag,
                    allVehicles[2].inPlatFlag,allVehicles[3].inPlatFlag);
        od

doneVehicles++;
do
:: (totVehicles == doneVehicles) -> break;
od

done:
        printf("Vehicle %d is done.\n",input_id);

}

/*-------------------------------------------------------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------------------------------------------------------*/
                        /*These are the verification variables.*/
/*-------------------------------------------------------------------------------------------------------------------------*/
#define phy1 (allVehicles[0].platDist == 0)
#define cp1 (allVehicles[0].posRecert == true)
#define br1 (allVehicles[0].brJoin == true)
#define gps1 (allVehicles[0].GPSOutput == true)
#define pj1 (allVehicles[0].inPlatFlag == true)

#define phy2 (allVehicles[1].platDist == 0)
#define cp2 (allVehicles[1].posRecert == true)
#define br2 (allVehicles[1].brJoin == true)
#define gps2 (allVehicles[1].GPSOutput == true)
#define pj2 (allVehicles[1].inPlatFlag == true)

#define phy3 (allVehicles[2].platDist == 0)
#define cp3 (allVehicles[2].posRecert == true)
#define br3 (allVehicles[2].brJoin == true)
#define gps3 (allVehicles[2].GPSOutput == true)
#define pj3 (allVehicles[2].inPlatFlag == true)

#define phy4 (allVehicles[3].platDist == 0)
#define cp4 (allVehicles[3].posRecert == true)
#define br4 (allVehicles[3].brJoin == true)
```

```
#define gps4 (allVehicles[3].GPSOutput == true)
#define pj4 (allVehicles[3].inPlatFlag == true)

#define phy5 (allVehicles[4].platDist == 0)
#define cp5 (allVehicles[4].posRecert == true)
#define br5 (allVehicles[4].brJoin == true)
#define gps5 (allVehicles[4].GPSOutput == true)
#define pj5 (allVehicles[4].inPlatFlag == true)
/*-------------------------------------------------------------------------------------------------------------------------*/
```

APPENDIX B.

SPIN MODEL CHECKER RESULTS

The input neverclaims and the output results from each run are listed here.

1. INVARIANT 1

```
never  {    /* <>(( (bm1 && pj1) && [] cp1) || ((bm2 && pj2) && [] cp2) || ((bm3 && pj3) && [] cp3)
    || ((bm4 && pj4) && [] cp4) || ((bm5 && pj5) && [] cp5) ) */
T0_init:
        do
        :: ((bm5 && pj5) && (cp5) && (((bm2 && pj2)) || ((((bm3 && pj3)) || ((((bm4 && pj4)) || ((bm5 && pj5)))))))
                && (((bm1 && pj1)) || ((((bm2 && pj2)) || ((((bm3 && pj3)) || ((((bm4 && pj4))
                || ((bm5 && pj5)))))))))) -> goto accept_S8
        :: ((bm4 && pj4) && (cp4) && (((bm2 && pj2)) || ((((bm3 && pj3))
                || ((((bm4 && pj4)) || ((bm5 && pj5)))))))) && (((bm1 && pj1)) || ((((bm2 && pj2)) || ((((bm3 && pj3))
                || ((((bm4 && pj4)) || ((bm5 && pj5)))))))))) -> goto accept_S13
        :: ((bm3 && pj3) && (cp3) && (((bm1 && pj1)) || ((((bm2 && pj2)) || ((((bm3 && pj3)) || ((((bm4 && pj4))
                || ((bm5 && pj5)))))))))) -> goto accept_S18
        :: ((bm2 && pj2) && (cp2)) -> goto accept_S23
        :: ((bm1 && pj1) && (cp1)) -> goto accept_S28
        :: (1) -> goto T0_init
        od;
accept_S8:
        do
        :: ((cp5)) -> goto accept_S8
        od;
accept_S13:
        do
        :: ((cp4)) -> goto accept_S13
        od;
accept_S18:
        do
        :: ((cp3)) -> goto accept_S18
        od;
accept_S23:
        do
        :: ((cp2)) -> goto accept_S23
        od;
accept_S28:
        do
        :: ((cp1)) -> goto accept_S28
        od;
}
```

Table 1. Results of SWARM Run with Invariant 1

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
816	753	204,456	28,413	232,869	96,415
816	725	243,886	36,332	280,218	102,936
808	735	28,381,118	38,064,635	66,445,753	12,814,668
808	729	30,575,598	41,628,799	72,204,397	13,935,548
816	127	170	61	231	115
816	127	171	65	236	116
808	127	598	632	1,230	362
808	127	676	777	1,453	411
816	753	191,892	28,091	219,983	93,636
816	725	189,763	29,578	219,341	90,567
808	725	2,183,192	3,151,575	5,334,767	1,076,571
808	729	582,218	848,611	1,430,829	280,930
816	753	45,593	7,912	53,505	29,672
816	725	243,742	36,322	280,064	102,931
808	751	30,783,886	43,674,842	74,458,728	15,912,275
808	729	24,976,745	32,844,658	57,821,403	10,447,175
808	847	29,291,801	39,621,779	68,913,580	15,361,838
816	127	171	65	236	116
808	127	674	711	1,385	401
808	127	597	602	1,199	358
808	847	17,972,675	26,314,383	44,287,058	11,849,226
816	725	212,117	32,513	244,630	94,837
808	743	4,463,935	6,497,292	10,961,227	2,336,744
808	743	4,463,935	6,497,292	10,961,227	2,336,744
808	813	636,057	952,454	1,588,511	416,629
816	725	41,486	6,990	48,476	24,977
816	747	183,383	24,031	207,414	98,344
808	737	30,467,168	41,334,789	71,801,957	13,637,976
808	853	33,886,973	48,881,998	82,768,971	21,379,071
808	729	23,777,078	31,012,443	54,789,521	9,309,718
816	127	170	61	231	115
808	127	654	690	1,344	406
808	127	614	594	1,208	365
808	729	16,075,896	22,989,194	39,065,090	8,251,318
816	747	142,540	19,738	162,278	77,083
808	731	2,110,392	3,030,938	5,141,330	988,954
808	853	1,224,382	1,825,033	3,049,415	840,614
808	729	293,910	428,211	722,121	134,435

2. INVARIANT 2

```
never  {    /* <>( (((!bm1 && pj1) U bc) && !cp1) || (((!bm2 && pj2) U bc) && !cp2) || (((!bm3 && pj3) U bc) && !cp3)
    || (((!bm4 && pj4) U bc) && !cp4) || (((!bm5 && pj5) U bc) && !cp5) ) */
T0_init:
        do
        :: atomic { (((! ((cp5)) && (bc) && ((! ((cp1))) || (((! ((cp2))) || (((! ((cp3))) || (((! ((cp4)))
            || (! ((cp5)))))))))) || (((! ((cp1)) && (bc)) || (((! ((cp2)) && (bc)) || (((! ((cp3)) && (bc))
            || (! ((cp4)) && (bc)))))))))) -> assert(!(((! ((cp5)) && (bc) && ((! ((cp1)))
            || (((! ((cp2))) || (((! ((cp3))) || (((! ((cp4))) || (! ((cp5)))))))))) || (((! ((cp1)) && (bc))
            || (((! ((cp2)) && (bc)) || (((! ((cp3)) && (bc)) || (! ((cp4)) && (bc)))))))))) }
        :: ((!bm5 && pj5) && ! ((cp5))) -> goto T0_S8
        :: ((!bm4 && pj4) && ! ((cp4))) -> goto T0_S13
        :: ((!bm3 && pj3) && ! ((cp3))) -> goto T0_S18
        :: ((!bm2 && pj2) && ! ((cp2))) -> goto T0_S23
        :: ((!bm1 && pj1) && ! ((cp1))) -> goto T0_S28
        :: (1) -> goto T0_init
        od;
T0_S8:
        do
        :: atomic { ((bc)) -> assert(!((bc))) }
        :: ((!bm5 && pj5)) -> goto T0_S8
        od;
T0_S13:
        do
        :: atomic { ((bc)) -> assert(!((bc))) }
        :: ((!bm4 && pj4)) -> goto T0_S13
        od;
T0_S18:
        do
        :: atomic { ((bc)) -> assert(!((bc))) }
        :: ((!bm3 && pj3)) -> goto T0_S18
        od;
T0_S23:
        do
        :: atomic { ((bc)) -> assert(!((bc))) }
        :: ((!bm2 && pj2)) -> goto T0_S23
        od;
T0_S28:
        do
        :: atomic { ((bc)) -> assert(!((bc))) }
        :: ((!bm1 && pj1)) -> goto T0_S28
        od;
accept_all:
        skip
}
```

Table 2. Results of SWARM Run with Invariant 2

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
816	753	206,234	28,852	235,086	96,858
816	725	243,886	36,332	280,218	102,936
808	735	27,835,093	37,204,258	65,039,351	11,859,718
808	729	30,142,487	40,900,190	71,042,677	13,487,221
816	127	170	61	231	115
816	127	171	65	236	116
808	127	598	632	1,230	362
808	127	676	777	1,453	411
816	753	194,212	28,409	222,621	94,950
816	725	185,115	28,336	213,451	88,537
808	731	2,040,773	2,907,532	4,948,305	923,998
808	729	564,765	818,463	1,383,228	254,014
816	753	51,945	9,576	61,521	37,536
816	725	243,752	36,314	280,066	102,932
808	749	31,322,410	44,573,391	75,895,801	16,748,527
808	729	25,007,268	32,921,095	57,928,363	10,347,399
808	847	29,126,813	39,274,580	68,401,393	15,032,858
816	127	171	65	236	116
808	127	674	711	1,385	401
808	127	597	602	1,199	358
808	851	17,971,120	26,335,139	44,306,259	12,212,903
816	725	214,758	32,977	247,735	97,961
808	743	4,407,125	6,397,217	10,804,342	2,301,951
808	725	1,122,424	1,627,964	2,750,388	544,426
808	813	628,355	935,005	1,563,360	385,539
816	725	42,449	7,330	49,779	26,317
816	747	183,383	24,031	207,414	98,344
808	737	30,817,322	41,965,956	72,783,278	14,163,782
808	853	34,269,927	49,543,720	83,813,647	21,927,472
808	729	25,481,552	33,934,582	59,416,134	11,325,940
816	127	170	61	231	115
808	127	654	690	1,344	406
808	127	614	594	1,208	365
808	729	15,836,338	22,547,093	38,383,431	8,236,033
816	747	155,747	21,849	177,596	86,278
808	735	2,202,870	3,190,277	5,393,147	1,126,535
808	853	1,226,534	1,822,242	3,048,776	757,909
808	729	275,013	395,867	670,880	129,934

3. INVARIANT 3

```
never  {    /* <>((phy1 && cp1 && br1 && gps1) && []!pj1) || <>((phy2 && cp2 && br2 && gps2) && []!pj2)
    || <>((phy3 && cp3 && br3 && gps3) && []!pj3)  || <>((phy4 && cp4 && br4 && gps4) && []!pj4)
    || <>((phy5 && cp5 && br5 && gps5) && []!pj5) */
T0_init:
        do
        :: (! ((pj5)) && (phy5 && cp5 && br5 && gps5)) -> goto accept_S8
        :: (! ((pj4)) && (phy4 && cp4 && br4 && gps4)) -> goto accept_S13
        :: (! ((pj3)) && (phy3 && cp3 && br3 && gps3)) -> goto accept_S18
        :: (! ((pj2)) && (phy2 && cp2 && br2 && gps2)) -> goto accept_S23
        :: (! ((pj1)) && (phy1 && cp1 && br1 && gps1)) -> goto accept_S28
        :: (1) -> goto T0_init
        od;
accept_S8:
        do
        :: (! ((pj5))) -> goto accept_S8
        od;
accept_S13:
        do
        :: (! ((pj4))) -> goto accept_S13
        od;
accept_S18:
        do
        :: (! ((pj3))) -> goto accept_S18
        od;
accept_S23:
        do
        :: (! ((pj2))) -> goto accept_S23
        od;
accept_S28:
        do
        :: (! ((pj1))) -> goto accept_S28
        od;
}
```

Table 3. Results of SWARM Run with Invariant 3

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
956	1,138	9,407,613	2,605,501	12,013,114	16,721,202
956	1,180	15,931,527	4,796,155	20,727,682	28,524,062
956	1,130	31,072,620	44,609,830	75,682,450	103,898,130
956	1,327	32,506,261	34,476,889	66,983,150	95,934,178
956	127	2,885	3,564	6,449	15,446
956	127	2,885	3,564	6,449	15,446
956	127	49,448	120,648	170,096	403,034
956	127	43,017	115,763	158,780	333,208
956	1,104	2,643,587	742,722	3,386,309	5,010,876
956	1,180	1,420,217	495,914	1,916,131	2,974,509
956	1,244	2,550,553	4,019,147	6,569,700	9,014,614
956	1,153	575,849	655,519	1,231,368	2,019,326
956	1,031	113,101	39,316	152,417	302,237
956	1,181	15,894,979	5,208,079	21,103,058	30,021,955
956	1,069	38,273,826	48,202,085	86,475,911	132,513,320
956	1,164	27,399,258	38,946,941	66,346,199	88,671,629
956	1,127	31,230,776	33,488,785	64,719,561	95,536,521
956	127	2,885	3,564	6,449	15,446
956	127	52,533	139,862	192,395	500,244
956	127	49,335	118,934	168,269	412,105
956	1,126	19,871,272	24,300,504	44,171,776	71,387,587
956	1,180	2,567,632	826,050	3,393,682	5,078,494
956	1,054	5,075,833	6,623,264	11,699,097	18,677,308
956	1,169	1,309,111	2,092,639	340,175	5,067,027
956	1,126	655,810	803,853	1,459,663	2,642,956
956	1,176	104,757	36,255	141,012	239,438
956	1,130	16,366,132	4,015,239	20,381,371	26,160,139
956	1,066	36,267,229	53,011,589	89,278,818	124,954,250
956	1,108	39,937,254	51,028,052	90,965,306	144,614,480
956	1,354	27,215,165	27,932,574	55,147,739	78,728,448
956	127	2,885	3,564	6,449	15,446
956	127	56,594	151,298	207,892	553,279
956	127	65,821	166,900	232,721	516,105
956	1,302	18,726,949	23,783,120	42,510,069	64,571,102
956	1,078	1,415,716	433,747	1,849,463	2,782,185
956	1,068	2,534,406	3,674,176	6,208,582	9,603,547
956	1,108	1,278,095	1,533,759	2,811,854	4,840,861
956	1,175	266,354	296,347	562,701	913,173

4. INVARIANT 4

```
/*spin -f '<>((phy1 && !cp1 && br1 && gps1) && []!pjr1) || <>((phy2 && !cp2 && br2 && gps2) && []!pjr2)
    || <>((phy3 && !cp3 && br3 && gps3) && []!pjr3)  || <>((phy4 && !cp4 && br4 && gps4) && []!pjr4)
    || <>((phy5 && !cp5 && br5 && gps5) && []!pjr5) '*/
never  {    /* <>((phy1 && !cp1 && br1 && gps1) && []!pjr1) || <>((phy2 && !cp2 && br2 && gps2) && []!pjr2)
    || <>((phy3 && !cp3 && br3 && gps3) && []!pjr3)  || <>((phy4 && !cp4 && br4 && gps4) && []!pjr4)
    || <>((phy5 && !cp5 && br5 && gps5) && []!pjr5)  */
T0_init:
        do
        :: (! ((pjr5)) && (phy5 && !cp5 && br5 && gps5)) -> goto accept_S8
        :: (! ((pjr4)) && (phy4 && !cp4 && br4 && gps4)) -> goto accept_S13
        :: (! ((pjr3)) && (phy3 && !cp3 && br3 && gps3)) -> goto accept_S18
        :: (! ((pjr2)) && (phy2 && !cp2 && br2 && gps2)) -> goto accept_S23
        :: (! ((pjr1)) && (phy1 && !cp1 && br1 && gps1)) -> goto accept_S28
        :: (1) -> goto T0_init
        od;
accept_S8:
        do
        :: (! ((pjr5))) -> goto accept_S8
        od;
accept_S13:
        do
        :: (! ((pjr4))) -> goto accept_S13
        od;
accept_S18:
        do
        :: (! ((pjr3))) -> goto accept_S18
        od;
accept_S23:
        do
        :: (! ((pjr2))) -> goto accept_S23
        od;
accept_S28:
        do
        :: (! ((pjr1))) -> goto accept_S28
        od;
}
```

Table 4. Results of SWARM Run with Invariant 4

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
956	1,141	10,218,064	7,207,217	17,425,281	27,310,073
956	1,233	15,559,424	5,548,570	21,107,994	20,408,911
956	1,175	31,880,045	54,827,844	86,707,889	63,217,105
956	1,332	32,254,401	94,124,619	126,379,020	148,953,090
956	127	1,711	1,556	3,267	7,410
956	127	1,711	1,556	3,267	7,410
956	127	25,490	48,420	73,910	184,383
956	127	23,837	48,239	72,076	149,404
956	1,116	2,747,971	2,002,195	4,750,166	7,473,328
956	1,233	1,285,067	502,562	1,787,629	1,658,695
956	1,281	2,544,452	4,505,703	7,050,155	5,848,261
956	1,263	577,641	1,622,202	2,199,843	2,654,252
956	1,066	101,205	76,090	177,295	323,158
956	1,233	14,776,948	5,284,942	20,061,890	19,924,264
956	1,179	39,750,370	113,580,300	153,330,680	197,395,510
956	1,170	27,522,286	46,747,016	74,269,302	43,800,791
956	1,533	31,565,179	91,500,946	123,066,120	150,650,430
956	127	1,711	1,556	3,267	7,410
956	127	27,404	58,181	85,585	265,846
956	127	26,640	50,761	77,401	191,002
956	1,533	20,437,157	59,495,185	79,932,342	100,470,560
956	1,235	2,675,951	1,043,583	3,719,534	3,959,385
956	1,122	5,216,659	10,081,655	15,298,314	17,734,810
956	1,183	1,317,947	2,441,155	3,759,102	2,968,017
956	1,533	676,576	1,889,386	2,565,962	3,466,116
956	1,233	105,177	40,697	145,874	199,468
956	1,116	16,038,915	12,197,944	28,236,859	40,411,013
956	1,132	36,521,300	66,481,238	103,002,540	71,224,071
956	1,515	41,073,948	117,936,830	159,010,780	211,184,030
956	1,357	25,689,990	74,486,647	100,176,640	110,045,800
956	127	1,711	1,556	3,267	7,410
956	127	29,068	60,370	89,438	288,188
956	127	35,182	64,282	99,464	231,623
956	1,337	18,452,374	54,591,848	73,044,222	85,265,369
956	1,087	1,481,761	1,072,906	2,554,667	3,913,248
956	1,141	2,655,055	4,908,177	7,563,232	6,707,586
956	1,515	1,335,918	3,900,147	5,236,065	6,433,955
956	1,281	310,844	896,289	1,207,133	1,697,366

5. INVARIANT 5

```
never  {    /* <>( !(pj1-> pj1 U !cp1) ||  !(pj2-> pj2 U !cp2) ||  !(pj3-> pj3 U !cp3) ||  !(pj4-> pj4 U !cp4)
    ||  !(pj5-> pj5 U !cp5) ) */
T0_init:
        do
        :: ((cp5) && (pj5)) -> goto accept_S8
        :: ((cp4) && (pj4)) -> goto accept_S13
        :: ((cp3) && (pj3)) -> goto accept_S18
        :: ((cp2) && (pj2)) -> goto accept_S23
        :: ((cp1) && (pj1)) -> goto accept_S28
        :: (1) -> goto T0_init
        od;
accept_S8:
        do
        :: ((cp5)) -> goto accept_S8
        :: atomic { (! ((pj5)) && (cp5)) -> assert(!(! ((pj5)) && (cp5))) }
        od;
accept_S13:
        do
        :: ((cp4)) -> goto accept_S13
        :: atomic { (! ((pj4)) && (cp4)) -> assert(!(! ((pj4)) && (cp4))) }
        od;
accept_S18:
        do
        :: ((cp3)) -> goto accept_S18
        :: atomic { (! ((pj3)) && (cp3)) -> assert(!(! ((pj3)) && (cp3))) }
        od;
accept_S23:
        do
        :: ((cp2)) -> goto accept_S23
        :: atomic { (! ((pj2)) && (cp2)) -> assert(!(! ((pj2)) && (cp2))) }
        od;
accept_S28:
        do
        :: ((cp1)) -> goto accept_S28
        :: atomic { (! ((pj1)) && (cp1)) -> assert(!(! ((pj1)) && (cp1))) }
        od;
accept_all:
        skip
}
```

Table 5. Results of SWARM Run with Invariant 5

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
816	753	1,274,385	1,069,459	2,343,844	1,031,633
816	725	1,464,240	1,572,176	3,036,416	1,132,014
808	735	27,430,912	36,443,322	63,874,234	11,418,194
808	729	30,093,469	40,888,082	70,981,551	13,343,598
816	127	1,010	1,346	2,356	1,115
816	127	1,016	1,395	2,411	1,126
808	127	3,632	8,253	11,885	3,476
808	127	4,046	9,692	13,738	3,661
816	753	643,829	102,184	746,013	353,400
816	725	173,924	26,551	200,475	79,501
808	727	2,250,438	3,269,106	5,519,544	1,126,388
808	729	588,910	862,431	1,451,341	293,054
816	753	42,562	7,282	49,844	26,891
816	725	1,463,937	1,539,085	3,003,022	1,130,723
808	749	29,984,968	42,215,632	72,200,600	14,807,840
808	729	25,302,028	33,464,744	58,766,772	10,583,126
808	847	29,711,279	40,313,547	70,024,826	15,439,919
816	127	1,016	1,395	2,411	1,126
808	127	3,981	8,967	12,948	3,592
808	127	3,644	8,279	11,923	3,303
808	851	17,779,821	25,916,686	43,696,507	11,481,513
816	729	897,549	432,836	1,330,385	64,7491
808	751	4,125,277	5,909,633	10,034,910	2,011,409
808	725	1,092,430	1,574,674	2,667,104	527,033
808	831	634,698	947,352	1,582,050	457,491
816	725	48,018	8,551	56,569	32,222
816	747	1,102,390	1,153,447	2,255,837	1,081,634
808	737	30,408,873	41,197,518	71,606,391	13,758,782
808	853	33,917,412	48,926,678	82,844,090	21,441,380
808	729	25,177,692	33,340,322	58,518,014	10,608,701
816	127	1,010	1,346	2,356	1,115
808	127	3,957	8,964	12,921	3,634
808	127	3,674	8,284	11,958	3,365
808	729	15,986,358	22,791,193	38,777,551	8,165,701
816	751	407,955	62,411	470,366	254,840
808	727	2,233,412	3,243,600	5,477,012	1,115,197
808	853	1,235,031	1,843,224	3,078,255	838,848
808	729	287,163	416,905	704,068	129,898

6. INVARIANT 6

```
/*spin -f '<>( (pjr1 && (!pjr1 V (cp1 && !br1))) || (pjr2 && (!pjr2 V (cp2 && !br2))) || (pjr3 &&
    (!pjr3 V (cp3 && !br3))) || (pjr4 && (!pjr4 V (cp4 && !br4))) || (pjr5 && (!pjr5 V (cp5 && !br5))))'*/
never  {    /* <>( (pjr1 && (!pjr1 V (cp1 && !br1))) || (pjr2 && (!pjr2 V (cp2 && !br2)))
    || (pjr3 && (!pjr3 V (cp3 && !br3))) || (pjr4 && (!pjr4 V (cp4 && !br4)))
    || (pjr5 && (!pjr5 V (cp5 && !br5))) ) */
T0_init:
        do
        :: ((cp5 && !br5) && (pjr5)) -> goto accept_S8
        :: ((cp4 && !br4) && (pjr4)) -> goto accept_S13
        :: ((cp3 && !br3) && (pjr3)) -> goto accept_S18
        :: ((cp2 && !br2) && (pjr2)) -> goto accept_S23
        :: ((cp1 && !br1) && (pjr1)) -> goto accept_S28
        :: (1) -> goto T0_init
        od;
accept_S8:
        do
        :: ((cp5 && !br5)) -> goto accept_S8
        :: atomic { (! ((pjr5)) && (cp5 && !br5)) -> assert(!(! ((pjr5)) && (cp5 && !br5))) }
        od;
accept_S13:
        do
        :: ((cp4 && !br4)) -> goto accept_S13
        :: atomic { (! ((pjr4)) && (cp4 && !br4)) -> assert(!(! ((pjr4)) && (cp4 && !br4))) }
        od;
accept_S18:
        do
        :: ((cp3 && !br3)) -> goto accept_S18
        :: atomic { (! ((pjr3)) && (cp3 && !br3)) -> assert(!(! ((pjr3)) && (cp3 && !br3))) }
        od;
accept_S23:
        do
        :: ((cp2 && !br2)) -> goto accept_S23
        :: atomic { (! ((pjr2)) && (cp2 && !br2)) -> assert(!(! ((pjr2)) && (cp2 && !br2))) }
        od;
accept_S28:
        do
        :: ((cp1 && !br1)) -> goto accept_S28
        :: atomic { (! ((pjr1)) && (cp1 && !br1)) -> assert(!(! ((pjr1)) && (cp1 && !br1))) }
        od;
accept_all:
        skip
}
```

Table 6. Results of SWARM Run with Invariant 6

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
816	753	204,456	28,413	232,869	96,415
816	725	243,886	36,332	280,218	102,936
808	735	28,381,118	38,064,635	66,445,753	12,814,668
808	729	30,575,598	41,628,799	72,204,397	13,935,548
816	127	170	61	231	115
816	127	171	65	236	116
808	127	598	632	1,230	362
808	127	676	777	1,453	411
816	753	191,892	28,091	219,983	93,636
816	725	189,763	29,578	219,341	90,567
808	725	2,183,192	3,151,575	5,334,767	1,076,571
808	729	582,218	848,611	1,430,829	280,930
816	753	45,593	7,912	53,505	29,672
816	725	243,742	36,322	280,064	102,931
808	751	30,783,886	43,674,842	74,458,728	15,912,275
808	729	24,976,745	32,844,658	57,821,403	10,447,175
808	847	29,291,801	39,621,779	68,913,580	15,361,838
816	127	171	65	236	116
808	127	674	711	1,385	401
808	127	597	602	1,199	358
808	847	17,972,675	26,314,383	44,287,058	11,849,226
816	725	212,117	32,513	244,630	94,837
808	743	4,463,935	6,497,292	10,961,227	2,336,744
808	725	1,117,658	1,619,542	2,737,200	520,850
808	813	636,057	952,454	1,588,511	416,629
816	725	41,486	6,990	48,476	24,977
816	747	183,383	24,031	207,414	98,344
808	737	30,467,168	41,334,789	71,801,957	13,637,976
808	853	33,886,973	48,881,998	82,768,971	21,379,071
808	729	23,777,078	31,012,443	54,789,521	9,309,718
816	127	170	61	231	115
808	127	654	690	1,344	406
808	127	614	594	1,208	365
808	729	16,075,896	22,989,194	39,065,090	8,251,318
816	747	142,540	19,738	162,278	77,083
808	731	2,110,392	3,030,938	5,141,330	988,954
808	853	1,224,382	1,825,033	3,049,415	840,614
808	729	293,910	428,211	722,121	134,435

7. INVARIANT 7

```
/*spin -f '<>( ((pj1 || pjr1) && []!nb1) ||  ((pj2 || pjr2) && []!nb2) || ((pj3 || pjr3) && []!nb3)
    || ((pj4 || pjr4) && []!nb4) || ((pj5 || pjr5) && []!nb5) )'*/
never  {    /* <>( ((pj1 || pjr1) && []!nb1) ||  ((pj2 || pjr2) && []!nb2) || ((pj3 || pjr3) && []!nb3)
    || ((pj4 || pjr4) && []!nb4) || ((pj5 || pjr5) && []!nb5) ) */
T0_init:
        do
        :: (! ((nb5)) && (pj5 || pjr5)) -> goto accept_S8
        :: (! ((nb4)) && (pj4 || pjr4)) -> goto accept_S13
        :: (! ((nb3)) && (pj3 || pjr3)) -> goto accept_S18
        :: (! ((nb2)) && (pj2 || pjr2)) -> goto accept_S23
        :: (! ((nb1)) && (pj1 || pjr1)) -> goto accept_S28
        :: (1) -> goto T0_init
        od;
accept_S8:
        do
        :: (! ((nb5))) -> goto accept_S8
        od;
accept_S13:
        do
        :: (! ((nb4))) -> goto accept_S13
        od;
accept_S18:
        do
        :: (! ((nb3))) -> goto accept_S18
        od;
accept_S23:
        do
        :: (! ((nb2))) -> goto accept_S23
        od;
accept_S28:
        do
        :: (! ((nb1))) -> goto accept_S28
        od;
}
```

Table 7. Results of SWARM Run with Invariant 7

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
816	753	1,116,427	936,131	2,052,558	1,022,076
816	725	1,263,187	1,314,700	2,577,887	1,112,634
808	712	27,797,605	35,532,631	63,330,236	12,902,621
808	700	30,204,975	39,504,081	69,709,056	14,215,258
816	127	1,010	1,346	2,356	1,115
816	127	1,016	1,395	2,411	1,126
808	127	3,632	8,253	11,885	3,476
808	127	4,046	9,692	13,738	3,661
816	724	341,015	48,650	389,665	179,352
816	696	284,434	42,712	327,146	151,229
808	700	2,359,063	3,266,542	5,625,605	1,381,344
808	700	576,152	774,053	1,350,205	276,787
816	724	49,018	8,622	57,640	33,596
816	725	1,265,548	1,291,277	2,556,825	1,112,106
808	596	25,672,493	36,579,298	62,251,791	34,045,296
808	700	24,750,687	31,076,926	55,827,613	10,890,658
808	820	29,713,006	38,896,014	68,609,020	16,742,644
816	127	1,016	1,395	2,411	1,126
808	127	3,981	8,967	12,948	3,592
808	127	3,644	8,279	11,923	3,303
808	820	17,683,950	24,934,412	42,618,362	11,990,964
816	700	475,556	71,686	547,242	243,440
808	716	4,229,425	5,758,692	9,988,117	2,281,402
808	704	1,131,222	1,531,036	2,662,258	625,161
808	790	611,954	873,998	1,485,952	353,098
816	696	56,446	10,766	67,212	43,120
816	747	968,283	990,557	1,958,840	1,070,151
808	578	27,264,717	37,658,936	64,923,653	34,830,403
808	824	33,551,519	46,785,044	80,336,563	21,464,601
808	700	23,174,232	28,856,972	52,031,204	9,339,704
816	127	1,010	1,346	2,356	1,115
808	127	3,957	8,964	12,921	3,634
808	127	3,674	8,284	11,958	3,365
808	700	14,366,058	19,298,375	33,664,433	6,806,685
816	720	239,374	33,017	272,391	140,727
808	704	2,274,409	3,131,654	5,406,063	1,212,087
808	824	1,235,328	1,766,064	3,001,392	873,162
808	704	301,929	408,549	710,478	139,172

8. INVARIANT 8

```
/*spin -f '<>( (nb1 && !bc) || (nb1 && !bc)  || (nb1 && !bc)  || (nb1 && !bc)  || (nb1 && !bc)  )'*/
never  {    /* <>( (nb1 && !bc) || (nb1 && !bc)  || (nb1 && !bc)  || (nb1 && !bc)  || (nb1 && !bc)  ) */
T0_init:
        do
        :: atomic { (( (nb1 && !bc) || (nb1 && !bc)  || (nb1 && !bc)  || (nb1 && !bc)  || (nb1 && !bc)  ))
            -> assert(!(( (nb1 && !bc) || (nb1 && !bc)  || (nb1 && !bc)  || (nb1 && !bc)
            || (nb1 && !bc)  ))) }
        :: (1) -> goto T0_init
        od;
accept_all:
        skip
}
```

Table 8. Results of SWARM Run with Invariant 8

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
816	753	204,939	28,889	233,828	96,895
816	725	243,886	36,332	280,218	102,936
808	739	27,505,430	36,583,677	64,089,107	11,675,708
808	729	30,921,101	42,190,526	73,111,627	14,510,660
816	127	170	61	231	115
816	127	171	65	236	116
808	127	598	632	1,230	362
808	127	676	777	1,453	411
816	753	140,265	19,997	160,262	64,091
816	725	155,408	23,541	178,949	70,940
808	727	2,162,448	3,115,806	5,278,254	1,018,411
808	729	538,809	774,334	1,313,143	234,872
816	753	45,533	8,238	53,771	31,020
816	725	243,853	36,322	280,175	102,914
808	749	30,455,657	43,050,727	73,506,384	15,303,546
808	729	24,990,344	32,828,963	57,819,307	10,201,987
808	847	29,160,278	39,280,399	68,440,677	14,622,920
816	127	171	65	236	116
808	127	674	711	1,385	401
808	127	597	602	1,199	358
808	847	17,755,394	25,842,837	43,598,231	11,208,394
816	725	212,659	32,381	245,040	94,685
808	759	4,333,691	6,261,263	10,594,954	2,188,190
808	725	1,120,969	1,623,742	2,744,711	545,256
808	819	625,606	932,878	1,558,484	408,906
816	725	47,229	8,474	55,703	30,309
816	747	183,383	24,031	207,414	98,344
808	737	30,776,081	41,863,200	72,639,281	13,939,558
808	853	33,617,258	48,366,502	81,983,760	20,451,865
808	729	25,161,891	33,381,888	58,543,779	10,788,141
816	127	170	61	231	115
808	127	654	690	1,344	406
808	127	614	594	1,208	365
808	729	16,162,016	23,121,266	39,283,282	8,350,128
816	747	162,974	22,883	185,857	90,083
808	729	2,243,229	3,266,434	5,509,663	1,189,533
808	853	1,217,636	1,810,031	3,027,667	830,502
808	729	305,856	450,791	756,647	160,549

9. INVARIANT 9

```
/*spin -f '<>( ((pj1 || pjr1) && (br1 V !pl1)) || ((pj2 || pjr2) && (br2 V !pl2))  || ((pj3 || pjr3)
    && (br3 V !pl3))  || ((pj4 || pjr4) && (br4 V !pl4))  || ((pj5 || pjr5) && (br5 V !pl5)) )'*/
never  {    /* <>( ((pj1 || pjr1) && (br1 V !pl1)) || ((pj2 || pjr2) && (br2 V !pl2))  || ((pj3 || pjr3)
    && (br3 V !pl3)) || ((pj4 || pjr4) && (br4 V !pl4))  || ((pj5 || pjr5) && (br5 V !pl5)) ) */
T0_init:
        do
        :: (! ((pl5)) && (pj5 || pjr5)) -> goto accept_S8
        :: atomic { (((! ((pl1)) && (br1) && (pj1 || pjr1)) || (((! ((pl2)) && (br2) && (pj2 || pjr2)) ||
            (((! ((pl3)) && (br3) && (pj3 || pjr3)) || (((! ((pl4)) && (br4) && (pj4 || pjr4))
            || (! ((pl5)) && (br5) && (pj5 || pjr5))))))))) -> assert(!(((! ((pl1)) && (br1) && (pj1 || pjr1))
            || (((! ((pl2)) && (br2) && (pj2 || pjr2)) || (((! ((pl3)) && (br3) && (pj3 || pjr3))
            || (((! ((pl4)) && (br4) && (pj4 || pjr4)) || (! ((pl5)) && (br5) && (pj5 || pjr5))))))))))) }
        :: (! ((pl4)) && (pj4 || pjr4)) -> goto accept_S13
        :: (! ((pl3)) && (pj3 || pjr3)) -> goto accept_S18
        :: (! ((pl2)) && (pj2 || pjr2)) -> goto accept_S23
        :: (! ((pl1)) && (pj1 || pjr1)) -> goto accept_S28
        :: (1) -> goto T0_init
        od;
accept_S8:
        do
        :: (! ((pl5))) -> goto accept_S8
        :: atomic { (! ((pl5)) && (br5)) -> assert(!(! ((pl5)) && (br5))) }
        od;
accept_S13:
        do
        :: (! ((pl4))) -> goto accept_S13
        :: atomic { (! ((pl4)) && (br4)) -> assert(!(! ((pl4)) && (br4))) }
        od;
accept_S18:
        do
        :: (! ((pl3))) -> goto accept_S18
        :: atomic { (! ((pl3)) && (br3)) -> assert(!(! ((pl3)) && (br3))) }
        od;
accept_S23:
        do
        :: (! ((pl2))) -> goto accept_S23
        :: atomic { (! ((pl2)) && (br2)) -> assert(!(! ((pl2)) && (br2))) }
        od;
accept_S28:
        do
        :: (! ((pl1))) -> goto accept_S28
        :: atomic { (! ((pl1)) && (br1)) -> assert(!(! ((pl1)) && (br1))) }
        od;
accept_all:
        skip
}
```

Table 9. Results of SWARM Run with Invariant 9

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
816	728	1,077,426	985,856	2,063,282	1,447,158
816	700	1,170,621	1,328,235	2,498,856	1,622,113
808	714	29,207,074	40,660,942	69,868,016	15,573,240
808	704	30,335,723	42,423,723	72,759,446	15,730,707
816	127	1,010	1,346	2,356	1,115
816	127	1,016	1,395	2,411	1,126
808	127	3,666	8,336	12,002	3,308
808	127	4,046	9,692	13,738	3,661
816	728	530,862	94,605	625,467	440,033
816	704	434,879	81,187	516,066	378,013
808	702	2,121,392	3,163,588	5,284,980	963,881
808	704	610,282	933,175	1,543,457	304,884
816	728	42,373	8,128	50,501	40,806
816	700	1,171,979	1,312,078	2,484,057	1,616,475
808	734	30,711,975	44,830,511	75,542,486	17,207,627
808	704	25,681,104	35,310,828	60,991,932	12,915,444
808	822	29,946,142	41,978,643	71,924,785	17,902,743
816	127	1,016	1,395	2,411	1,126
808	127	3,907	8,791	12,698	3,499
808	127	3,545	7,733	11,278	3,176
808	826	18,315,380	27,639,395	45,954,775	12,306,180
816	702	553,931	105,720	659,651	455,540
808	722	4,528,594	6,810,295	11,338,889	2,316,875
808	702	1,163,965	1,769,539	2,933,504	560,815
808	798	617,175	948,765	1,565,940	329,662
816	708	60,419	12,988	73,407	64,962
816	722	912,863	995,392	1,908,255	1,397,736
808	712	30,537,285	42,654,330	73,191,615	15,780,654
808	828	34,392,902	51,159,710	85,552,612	22,123,435
808	704	25,462,663	34,906,570	60,369,233	12,874,215
816	127	1,010	1,346	2,356	1,115
808	127	3,978	8,968	12,946	3,655
808	127	3,674	8,284	11,958	3,365
808	704	14,577,557	21,100,918	35,678,475	7,474,534
816	722	388,526	65,470	453,996	339,177
808	714	2,208,872	3,319,485	5,528,357	1,068,086
808	828	1,245,536	1,922,889	3,168,425	754,519
808	704	293,584	444,361	737,945	127,730

10. INVARIANT 10

```
/*spin -f '<>( (!cp1 && (command1 || evaluate1)) || (!cp2 && (command2 || evaluate2))
    || (!cp3 && (command3 || evaluate3)) || (!cp4 && (command4 || evaluate4)) || (!cp5 &&
    (command5 || evaluate5)) )'*/
never  {    /* <>( (!cp1 && (command1 || evaluate1)) || (!cp2 && (command2 || evaluate2)) || (!cp3 &&
    (command3 || evaluate3)) || (!cp4 && (command4 || evaluate4)) || (!cp5 && (command5 || evaluate5)) ) */
T0_init:
        do
        :: atomic { (( (!cp1 && (command1 || evaluate1)) || (!cp2 && (command2 || evaluate2))
            || (!cp3 && (command3 || evaluate3)) || (!cp4 && (command4 || evaluate4)) ||
            (!cp5 && (command5 || evaluate5)) ))
            -> assert(!(( (!cp1 && (command1 || evaluate1)) || (!cp2 && (command2 || evaluate2))
            || (!cp3 && (command3 || evaluate3)) || (!cp4 && (command4 || evaluate4)) ||
            (!cp5 && (command5 || evaluate5)) ))) }
        :: (1) -> goto T0_init
        od;
accept_all:
        skip
}
```

Table 10. Results of SWARM Run with Invariant 10

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
816	753	205,260	28,561	233,821	95,970
816	725	243,886	36,332	280,218	102,936
808	735	27,505,538	36,529,428	64,034,966	11,653,494
808	729	30,930,729	42,204,840	73,135,569	14,320,936
816	127	170	61	231	115
816	127	171	65	236	116
808	127	598	632	1,230	362
808	127	676	777	1,453	411
816	753	194,019	28,189	222,208	93,716
816	725	196,509	30,347	226,856	94,638
808	725	2,285,098	3,322,785	5,607,883	1,144,529
808	729	578,081	839,950	1,418,031	266,247
816	753	37,345	6,238	43,583	22,139
816	725	243,853	36,330	280,183	102,936
808	751	30,764,256	43,604,076	74,368,332	15,908,577
808	729	24,813,298	32,555,581	57,368,879	10,089,344
808	847	29,222,183	39,455,101	68,677,284	15,323,503
816	127	171	65	236	116
808	127	674	711	1,385	401
808	127	597	602	1,199	358
808	851	17,677,424	25,720,185	43,397,609	11,074,941
816	725	220,897	34,009	254,906	100,152
808	751	4,325,805	6,240,264	10,566,069	2,075,032
808	725	1,151,403	1,672,028	2,823,431	531,519
808	813	622,957	923,595	1,546,552	392,054
816	725	49,636	8,916	58,552	33,204
816	747	183,383	24,031	207,414	98,344
808	737	30,980,444	42,225,314	73,205,758	14,404,424
808	853	34,140,561	49,333,531	83,474,092	21,667,323
808	729	25,529,617	33,970,646	59,500,263	11,168,421
816	127	170	61	231	115
808	127	654	690	1,344	406
808	127	614	594	1,208	365
808	729	15,483,597	21,959,991	37,443,588	7,472,546
816	747	164,660	23,043	187,703	90,920
808	735	2,165,477	3,129,449	5,294,926	1,053,003
808	853	1,230,499	1,833,314	3,063,813	822,381
808	729	308,148	453,576	761,724	155,684

11. INVARIANT 11

```
/*spin -f '<>( (pl1 && (phy1 || !brl1))  || (pl2 && (phy2 || !brl2))  || (pl3 && (phy3 || !brl3))
    || (pl4 && (phy4 || !brl4)) || (pl5 && (phy5 || !brl5)))'*/
never  {    /* <>( (pl1 && (phy1 || !brl1))  || (pl2 && (phy2 || !brl2))  || (pl3 && (phy3 || !brl3))
    || (pl4 && (phy4 || !brl4)) || (pl5 && (phy5 || !brl5))) */
T0_init:
        do
        :: atomic { (( (pl1 && (phy1 || !brl1))  || (pl2 && (phy2 || !brl2))  || (pl3 && (phy3 || !brl3))
            || (pl4 && (phy4 || !brl4)) || (pl5 && (phy5 || !brl5)))) -> assert(!(( (pl1 && (phy1 || !brl1))
            || (pl2 && (phy2 || !brl2))  || (pl3 && (phy3 || !brl3))  || (pl4 && (phy4 || !brl4))
            || (pl5 && (phy5 || !brl5))))) }
        :: (1) -> goto T0_init
        od;
accept_all:
        skip
}
```

Table 11. Results of SWARM Run with Invariant 11

State-Vector (bytes)	Depth Reached	States, Stored	States, Matched	Transitions	Atomic Steps
792	387	266,685	58,786	325,471	644,856
792	397	332,192	83,450	415,642	844,717
792	425	3,898,074	5,765,283	9,663,357	18,327,828
792	557	391,423	5,819,978	9,734,213	18,586,665
792	127	2,080	1,676	3,756	9,200
792	127	2,138	1,796	3,934	8,302
792	127	15,878	17,678	33,556	66,481
792	127	17,300	20,784	38,084	58,599
792	387	243,650	53,861	297,511	589,222
792	397	258,029	63,389	321,418	630,965
792	409	1,519,241	2,365,701	3,884,942	7,622,567
792	438	313,911	471,439	785,350	1,662,781
792	371	54,231	12,871	67,102	135,228
792	397	332,189	83,441	415,630	844,520
792	366	3,726,214	5,536,873	9,263,087	17,546,144
792	413	3,905,803	5,775,176	9,680,976	18,351,573
792	383	3,825,144	5,570,371	9,395,515	17,789,643
792	127	2,138	1,796	3,934	8,302
792	127	19,038	22,948	41,986	92,616
792	127	16,000	17,751	33,751	68,340
792	377	3,471,629	5,163,201	8,634,830	16,512,492
792	397	289,183	72,072	361,255	724,546
792	373	1,788,433	2,682,108	4,470,541	8,540,646
792	388	651,618	1,015,292	1,666,910	3,281,283
792	363	528,246	855,275	1,383,521	2,879,333
792	397	42,689	10,129	52,818	99,901
792	387	283,548	62,563	346,111	686,228
792	377	3,834,827	5,636,928	9,471,755	17,836,212
792	379	3,615,779	5,314,065	8,929,844	16,979,534
792	557	3,939,995	5,851,983	9,791,978	18,682,676
792	127	2,080	1,676	3,756	9,200
792	127	19,288	22,903	42,191	96,497
792	127	20,132	21,349	41,481	76,718
792	535	3,393,736	5,093,582	8,487,318	16,309,086
792	387	226,722	50,130	276,852	545,867
792	357	678,292	999,431	1,677,723	3,244,271
792	357	903,528	1,444,102	2,347,630	4,707,467
792	426	211,099	308,266	519,365	1,136,067

REFERENCES

Alexiou, N., Laganà, M., Gisdakis, S., Khodaei, M., and Papadimitratos, P., 'Vespa: Vehicular security and privacy-preserving architecture,' in 'Proceedings of the 2nd ACM workshop on hot topics on wireless network security and privacy,' ACM, 2013 pp. 19–24.

Alibaba.com, 'Cost of Antminer S9j,' https://www.alibaba.com/product-detail/Free-shipping-Antminer-S9j-14-5TH_60765088857.html, 2019.

Amoozadeh, M., Deng, H., Chuah, C.-N., Zhang, H. M., and Ghosal, D., 'Platoon management with cooperative adaptive cruise control enabled by VANET,' Vehicular communications, 2015, **2**(2), pp. 110–123.

Apostolaki, M., Zohar, A., and Vanbever, L., 'Hijacking bitcoin: Routing attacks on cryptocurrencies,' in '2017 IEEE Symposium on Security and Privacy (SP),' IEEE, 2017 pp. 375–392.

Bentov, I., Gabizon, A., and Mizrahi, A., 'Cryptocurrencies without proof of work,' in 'International Conference on Financial Cryptography and Data Security,' Springer, 2016 pp. 142–157.

Biryukov, A. and Pustogarov, I., 'Bitcoin over TOR isn't a good idea,' in '2015 IEEE Symposium on Security and Privacy,' IEEE, 2015 pp. 122–134.

BitInfoCharts, 'Bitcoin Hashrate chart,' https://bitinfocharts.com/comparison/bitcoin-hashrate.html, 2019a, accessed: May, 2019.

BitInfoCharts, 'Bitcoin SV Hashrate chart,' https://bitinfocharts.com/comparison/bitcoin sv-hashrate.html, 2019b, accessed: May, 2019.

BitInfoCharts, 'Litecoin Hashrate chart,' https://bitinfocharts.com/comparison/litecoin-hashrate.html, 2019c, accessed: May, 2019.

Blockchain.com, 'Total Hash Rate,' http://www.blockchain.com/en/charts/hash-rate, 2019.

Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J. A., and Felten, E. W., 'Sok: Research perspectives and challenges for bitcoin and cryptocurrencies,' in '2015 IEEE Symposium on Security and Privacy,' IEEE, 2015 pp. 104–121.

Borge, M., Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., and Ford, B., 'Proof-of-personhood: Redemocratizing permissionless cryptocurrencies,' in '2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW),' IEEE, 2017 pp. 23–26.

Bureau, C., *Statistical Abstract of the United States 2010*, Government Printing Office, 2009.

BuriedOne, 'GPU Mining Hashrates,' http://www.buriedone.com/gpu-hashrates, 2019.

Buterin, V. *et al.*, 'A next-generation smart contract and decentralized application platform,' white paper, 2014.

Castro, M., Liskov, B., *et al.*, 'Practical Byzantine fault tolerance,' in 'OSDI,' volume 99, 1999 pp. 173–186.

Chen, L., Lit, Q., Martin, K. M., and Ng, S.-L., 'A privacy-aware reputation-based announcement scheme for VANETs,' in '2013 IEEE 5th International Symposium on Wireless Vehicular Communications (WiVeC),' IEEE, 2013 pp. 1–5.

Christidis, K. and Devetsikiotis, M., 'Blockchains and smart contracts for the internet of things,' Ieee Access, 2016, **4**, pp. 2292–2303.

Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P., 'Model checking and the state explosion problem,' in 'LASER Summer School on Software Engineering,' Springer, 2011 pp. 1–30.

CoinMarketCap, 'Cryptocurrency Market Capitalizations,' https://coinmarketcap.com/, 2019.

Crosby, M., Pattanayak, P., Verma, S., Kalyanaraman, V., *et al.*, 'Blockchain technology: Beyond bitcoin,' Applied Innovation, 2016, **2**(6-10), p. 71.

Dorri, A., Steger, M., Kanhere, S. S., and Jurdak, R., 'Blockchain: A distributed solution to automotive security and privacy,' IEEE Communications Magazine, 2017, **55**(12), pp. 119–125.

Duan, J. and Chow, M.-Y., 'A resilient consensus-based distributed energy management algorithm against data integrity attacks,' IEEE Transactions on Smart Grid, 2018, **10**(5), pp. 4729–4740.

Dwork, C., Lynch, N., and Stockmeyer, L., 'Consensus in the presence of partial synchrony,' Journal of the ACM (JACM), 1988, **35**(2), pp. 288–323.

Elsadig, M. A. and Fadlalla, Y. A., 'VANETs security issues and challenges: a survey,' Indian Journal of Science and Technology, 2016, **9**(28), pp. 1–8.

Eyal, I., 'The miner's dilemma,' in '2015 IEEE Symposium on Security and Privacy,' IEEE, 2015 pp. 89–103.

Gao, F., Zhu, L., Shen, M., Sharif, K., Wan, Z., and Ren, K., 'A blockchain-based privacy-preserving payment mechanism for vehicle-to-grid networks,' IEEE network, 2018, **32**(6), pp. 184–192.

Gervais, A., Ritzdorf, H., Karame, G. O., and Capkun, S., 'Tampering with the delivery of blocks and transactions in bitcoin,' in 'Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security,' ACM, 2015 pp. 692–705.

Goyal, P., Parmar, V., and Rishi, R., 'MANET: vulnerabilities, challenges, attacks, application,' IJCEM International Journal of Computational Engineering & Management, 2011, **11**(2011), pp. 32–37.

Guette, G. and Bryce, C., 'Using TPMS to secure vehicular ad-hoc networks (VANETs),' in 'IFIP International Workshop on Information Security Theory and Practices,' Springer, 2008 pp. 106–116.

Gurung, S., Lin, D., Squicciarini, A., and Bertino, E., 'Information-oriented trustworthiness evaluation in vehicular ad-hoc networks,' in 'International Conference on Network and System Security,' Springer, 2013 pp. 94–108.

Hanson, M. and Uy, M. R., 'Best mining GPU 2020: the best graphics cards for mining Bitcoin, Ethereum and more,' http://www.techradar.com/news/best-mining-gpu, 2020.

Hartenstein, H. and Laberteaux, L., 'A tutorial survey on vehicular ad hoc networks,' IEEE Communications magazine, 2008, **46**(6), pp. 164–171.

Holzmann, G. J., *The SPIN model checker: Primer and reference manual*, volume 1003, Addison-Wesley Reading, 2004.

Holzmann, G. J., Joshi, R., and Groce, A., 'Swarm verification techniques,' IEEE Transactions on Software Engineering, 2010, **37**(6), pp. 845–857.

Howser, G. and McMillin, B., 'A modal model of stuxnet attacks on cyber-physical systems: A matter of trust,' in '2014 Eighth International Conference on Software Security and Reliability (SERE),' IEEE, 2014 pp. 225–234.

Infomation Technology Laboratory, N. I. o. S. and Technology, 'Federal Information Processing Standards Publication, Digitial Signature Standard (DSS),' https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf, 2013.

Jaimes, L. M. S., Ullah, K., and dos Santos Moreira, E., 'ARS: anonymous reputation system for vehicular ad hoc networks,' in '2016 8th IEEE Latin-American Conference on Communications (LATINCOM),' IEEE, 2016 pp. 1–6.

Jia, D., Lu, K., Wang, J., Zhang, X., and Shen, X., 'A survey on platoon-based vehicular cyber-physical systems,' IEEE communications surveys & tutorials, 2015, **18**(1), pp. 263–284.

Johnson, D., Menezes, A., and Vanstone, S., 'The elliptic curve digital signature algorithm (ECDSA),' International journal of information security, 2001, **1**(1), pp. 36–63.

Kamat, P., Baliga, A., and Trappe, W., 'An identity-based security framework for VANETs,' in 'Proceedings of the 3rd international workshop on Vehicular ad hoc networks,' ACM, 2006 pp. 94–95.

Kanteti, U. G., 'Multiple security domain model of a vehicle in an automated vehicle system,' 2017.

Kiayias, A. and Panagiotakos, G., 'Speed-Security Tradeoffs in Blockchain Protocols.' IACR Cryptology ePrint Archive, 2015, **2015**, p. 1019.

Lee, E. A., 'Cyber physical systems: Design challenges,' in '2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC),' IEEE, 2008 pp. 363–369.

Leiding, B., Memarmoshrefi, P., and Hogrefe, D., 'Self-managed and blockchain-based vehicular ad-hoc networks,' in 'Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct,' ACM, 2016 pp. 137–140.

Lu, Z., Wang, Q., Qu, G., and Liu, Z., 'Bars: a blockchain-based anonymous reputation system for trust management in VANETs,' in '2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE),' IEEE, 2018 pp. 98–103.

Maxwell, G., Poelstra, A., Seurin, Y., and Wuille, P., 'Simple Schnorr multi-signatures with applications to bitcoin,' Designs, Codes and Cryptography, 2018, pp. 1–26.

Moteff, J., Copeland, C., and Fischer, J., 'Critical infrastructures: What makes an infrastructure critical?' Library of Congress Washington DC Congressional Research Service, 2003 .

Nakamoto, S. *et al.*, 'Bitcoin: A peer-to-peer electronic cash system,' 2008.

Palaniswamy, P. and McMillin, B., 'Cyber-physical security of an electric microgrid,' in '2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC),' IEEE, 2018 pp. 74–83.

Pilkington, M., '11 Blockchain technology: principles and applications,' Research handbook on digital transformations, 2016, **225**.

Qu, F., Wu, Z., Wang, F.-Y., and Cho, W., 'A security and privacy review of VANETs,' IEEE Transactions on Intelligent Transportation Systems, 2015, **16**(6), pp. 2985–2996.

Reference, B. D., 'Bitcoin Block Headers,' https://bitcoin.org/en/developer-reference#block-headers, 2019.

Reis, A. B., Sargento, S., Neves, F., and Tonguz, O. K., 'Deploying roadside units in sparse vehicular networks: What really works and what does not,' IEEE transactions on vehicular technology, 2013, **63**(6), pp. 2794–2806.

Rodgers, T., 'Ethereum Classic Price Roaring Just Weeks After 51% Attack,' https://www.forbes.com/sites/tomrodgers1/2019/04/08/ethereum-classic-price-roaring-just-weeks-after-51-attack/#7677748b6f7e, 2019.

Roth, T. and McMillin, B., 'Physical attestation of cyber processes in the smart grid,' in 'International Workshop on Critical Information Infrastructures Security,' Springer, 2013 pp. 96–107.

Roth, T. P., 'Distributed state verification in the smart grid using physical attestation,' 2015.

Roy, A. and Madria, S., 'Distributed Incentive-Based Secured Traffic Monitoring in VANETs,' in '21st IEEE International Conference on Mobile Data Management,' IEEE, 2020 .

Sanseverino, E. R., Di Silvestre, M. L., Gallo, P., Zizzo, G., and Ippolito, M., 'The blockchain in microgrids for transacting energy and attributing losses,' in '2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData),' IEEE, 2017 pp. 925–930.

Sharma, P. K., Moon, S. Y., and Park, J. H., 'Block-VN: A Distributed Blockchain Based Vehicular Network Architecture in Smart City.' JIPS, 2017, **13**(1), pp. 184–195.

Singh, K., Saini, P., Rani, S., and Singh, A. K., 'Authentication and privacy preserving message transfer scheme for vehicular ad hoc networks (VANETs),' in 'Proceedings of the 12th ACM International Conference on Computing Frontiers,' ACM, 2015 p. 58.

Singh, M. and Kim, S., 'Blockchain based intelligent vehicle data sharing framework,' arXiv preprint arXiv:1708.09721, 2017.

Singh, M. and Kim, S., 'Branch based blockchain technology in intelligent vehicle,' Computer Networks, 2018, **145**, pp. 219–231.

Squicciarini, A., Lin, D., and Mancarella, A., 'Paim: Peer-based automobile identity management in vehicular ad-hoc network,' in '2011 IEEE 35th Annual Computer Software and Applications Conference,' IEEE, 2011 pp. 263–272.

Studer, A., Shi, E., Bai, F., and Perrig, A., 'TACKing together efficient authentication, revocation, and privacy in VANETs,' in '2009 6th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks,' IEEE, 2009 pp. 1–9.

Tschorsch, F. and Scheuermann, B., 'Bitcoin and beyond: A technical survey on decentralized digital currencies,' IEEE Communications Surveys & Tutorials, 2016, **18**(3), pp. 2084–2123.

Value, A. M., 'Bitmain Antminer L3 (596Mh) profitability: ASIC Miner Value,' http://www.asicminervalue.com/miners/bitmain/antminer-l3-596mh, 2019.

Wagner, D. and Schweitzer, B., 'The growing threat of cyber-attacks on critical infrastructure,' Huffington Post, 2016.

Wagner, M. and McMillin, B., 'Cyber-Physical Transactions: A Method for Securing VANETs with Blockchains,' in '2018 IEEE 23rd Pacific Rim International Symposium on Dependable Computing (PRDC),' 2018 pp. 64–73, doi:10.1109/PRDC.2018.00017.

Walker, J., 'Intelligent Transportation Systems - Security Credential Management System (SCMS),' http://www.its.dot.gov/resources/scms.htm, 2017.

Wang, W., Hoang, D. T., Xiong, Z., Niyato, D., Wang, P., Hu, P., and Wen, Y., 'A survey on consensus mechanisms and mining management in blockchain networks,' arXiv preprint arXiv:1805.02707, 2018, pp. 1–33.

Wiki, B., 'Miner fees,' https://en.bitcoin.it/wiki/Miner_fees, 2019.

Wright, James, Garrett, Kyle, J., Hill, Krueger, Gregory, Evans, H., J., Andrews, and et al., 'National Connected Vehicle Field Infrastructure Footprint Analysis,' 2014.

Xie, L., Ding, Y., Yang, H., and Wang, X., 'Blockchain-Based Secure and Trustworthy Internet of Things in SDN-Enabled 5G-VANETs,' IEEE Access, 2019, **7**, pp. 56656–56666.

Xie, Y., Wu, L., Zhang, Y., and Shen, J., 'Efficient and secure authentication scheme with conditional privacy-preserving for VANETs,' Chinese Journal of Electronics, 2016, **25**(5), pp. 950–956.

Xu, X., Weber, I., Staples, M., Zhu, L., Bosch, J., Bass, L., Pautasso, C., and Rimba, P., 'A taxonomy of blockchain-based systems for architecture design,' in '2017 IEEE International Conference on Software Architecture (ICSA),' IEEE, 2017 pp. 243–252.

Yang, Y.-T., Chou, L.-D., Tseng, C.-W., Tseng, F.-H., and Liu, C.-C., 'Blockchain-Based Traffic Event Validation and Trust Verification for VANETs,' IEEE Access, 2019, **7**, pp. 30868–30877.

Yuan, Y. and Wang, F.-Y., 'Towards blockchain-based intelligent transportation systems,' in '2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC),' IEEE, 2016 pp. 2663–2668.

Zeadally, S., Hunt, R., Chen, Y.-S., Irwin, A., and Hassan, A., 'Vehicular ad hoc networks (VANETs): status, results, and challenges,' Telecommunication Systems, 2012, **50**(4), pp. 217–241.

VITA

Matthew Edward Wagner was born in Dover, Delaware. He graduated from Lindenwood University in St. Charles, Missouri in 2016 with a Bachelor of Science in Computer Science and Mathematics with honors. Matthew received the Chancellors Distinguished Fellowship to attend Missouri University of Science and Technology. During his time at Missouri S&T, he taught one course on Algorithms and researched a variety of topics including cyber-physical systems, blockchains, vehicular ad-hoc networks, security, and privacy. He received his Master of Science degree in Computer Science in May 2019 and his Doctor of Philosophy in Computer Science in August 2020 from Missouri University of Science and Technology under Professor Bruce McMillin.