COUNTER AUTONOMY DEFENSE FOR AERIAL AUTONOMOUS SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Mark E. Duntz

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Aeronautics and Astronautics

May 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF THESIS APPROVAL

To my wife, Lexis, for whose infinite support I will be eternally grateful.

ACKNOWLEDGMENTS

Special thanks to Dr. Shaoshuai Mou for the mentorship and guidance he provided to me in the production of this thesis and to all of the fellow members of AIMS Lab for the help and support.

Additionally, I am especially thankful for the assistance Colonel (Ret.) Dave Hankins, USAF provided me throughout this journey. It is because of his help I was able to succeed at Purdue.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Duntz, Mark E. MSAAE, Purdue University, May 2020. Counter Autonomy Defense for Aerial Autonomous Systems. Major Professor: Shaoshuai Mou.

Here, we explore methods of counter autonomy defense for aerial autonomous multi-agent systems. First, the case is made for vast capabilities made possible by these systems. Recognizing that widespread use is likely on the horizon, we assert that it will be necessary for system designers to give appropriate attention to the security and vulnerabilities of such systems. We propose a method of learning-based resilient control for the multi-agent formation tracking problem, which uses reinforcement learning and neural networks to attenuate adversarial inputs and ensure proper operation. We also devise a learning-based method of cyber-physical attack detection for UAVs, which requires no formal system dynamics model yet learns to recognize abnormal behavior. We also utilize similar techniques for time signal analysis to achieve epileptic seizure prediction. Finally, a blockchain-based method for network security in the presence of Byzantine agents is explored.

1. INTRODUCTION

The unmanned aerial vehicle (UAV) has a long history tracing back to the first World War, before the advent of modern robotics and high performance computing. From the beginning, the chief motivation behind the technology was to reduce human risk while still accomplishing missions requiring the use of aircraft. Since then, incredible technological improvements in the areas of robotics, processing power, communications, and manufacturing have given life to a new era in aviation. In war, UAVs provide greater endurance and persistence while also reducing human risk. This gives combatant commanders unique capabilities that add flexibility and versatility to their arsenal. The United States military took early notice of this and was a key early developer of the technology, investing in platforms like the RQ-11 Raven, MQ-1 Predator, and the RQ-4 Global Hawk. Each of these platforms is drastically different in terms of size, cost, and role, yet equally important to the warfighter.

To a certain extent, a platform's combat role determines the degree of human operation required. For example, strike missions are carried out by human pilots operating the UAV via relayed control. Still, they all are capable of some level of autonomy, or exhibiting behavior and performing tasks without the need for dedicated operator attention. This reduces the operator workload and often decreases operating costs and increases effectiveness. It is expected that autonomy will only continue to mature, with the ultimate goal of autonomous execution with minimal high-level human guidance and direction. The US Air Force has plans for a loyal wingman platform, which replace conventional manned aircraft in favor of similarly-capable unmanned autonomous ones. The result would be a single pilot leading a formation of aircraft that could achieve effects which normally require a group of manned ones. Successful development and employment of such technology will mark a paradigm shift for Air Force aviators, as their role will expand to include multi-agent system management on top of the demanding responsibilities of operating a single ship. The Department of Defense also has aspirations of fielding fully-autonomous groups of low-cost air vehicles for certain missions. In 2016, US Navy F-18s successfully deployed a swarm of autonomous Perdix drones during a flight test at Naval Air Weapons Station China Lake. The test demonstrated key capabilities such as non-traditional deployment methods, waypoint following, and congregation in a large micro-UAV swarm. The potential applications for systems like this extends to intelligence, surveillance, and reconnaissance (ISR) gathering, strike, combat search-and-rescue, hazardous material detection, and projectile defense.

The Department of Defense is not the only entity interested in UAV development. Commercial organizations like Uber, Walmart, and UPS have taken stakes in the field motivated by the promises of quicker direct point-to-point transportation and delivery. In fact, the FAA projects a 300% increase in commercial UAV use by 2023 after having observed accelerated growth in previous years [1]. Aspirations of industry include utilizing UAVs for efficient people and cargo flow as well as remote sensing. Ultimately, a high-degree of autonomy is desired to reduce costs and improve operations efficiency.

One of the most attractive uses for UAVs in both the defense and commercial sectors is in the form of a multi-agent system. These have become wildly popular in recent years due to their ability to produce complex effects greater than the capabilities of their individual agents. The utility of their emergent behavior hinges upon the system's robustness and scalability, since interaction in distributed multi-agent systems occurs only locally without a central authority. Appropriately, interest in multi-agent systems as well as the supporting research stems from the broad application potential of the field including cooperative control over large, complex networks of agents.

Systems of autonomous air vehicles crowding the sky above us seems inevitable, so it is important to consider the areas of potential risk before that time comes. Due to their unique roles and missions, failures of these systems are particularly dangerous.

Consider the potential crises stemming from failures of UAVs used for urban air mobility. Similarly, vehicles in a combat zone could be armed with formidable weapons, and the effects of failure could be devastating. As with any innovative technology, adversaries will seek to exploit its vulnerabilities with nefarious intent. Especially in a military context, it is critical that system designers develop the techniques necessary to prevent this. For this reason, this work is concerned with improving these techniques. In particular, it addresses methods of resilient control and attack detection in the face of adversaries. Along the way, a hybrid method for time-signal analysis was developed and its applications were explored in another field based on a collaboration request from a partnering institution. For this reason, epileptic seizure analysis is performed using a method which could be just as effectively applied in the analysis of signals for autonomous system threat detection and defense. Additional work is also included on evaluating the potential of using blockchain technology to mitigate the effects of malicious/faulty information flow between agents.

1.1 Related Works

1.1.1 Multi-agent Systems

Multi-agent systems have become wildly popular in recent years. Originating out of inspiration from nature and biology, these systems concern decentralized networks of agents that swarm to produce effects greater than the sum of their parts [2–4]. Attractive features of such systems that stem from their distributed nature include robustness and scalability. Swarms are considered decentralized because global effects and behaviors are achieved by agents usually only interacting locally with their neighbors. No central authority plans the network's behavior and, thus, no single point of failure exists. Research in robotic swarms has been increasing drastically in recent years. As the cost of small robotic hardware decreases and computational power density increases, swarms become a more viable and smarter choice for many industrial applications, particularly in aerospace applications. Purpose-built, sophisticated aerospace vehicles are extremely costly. Highly adaptable systems that can be mass-produced and work together to accomplish the same tasks could be cheaper and more efficient. [5] acknowledges that recent developments in both hardware and swarm research suggest commercial adoption of drone swarms is near. Potential applications include package delivery, factory automation, air traffic management, on-demand sensing services, transportation, and search and rescue [6–13]. Still, challenges exist for organizations attempting to field and utilize robot swarms at the moment. For instance, secure communication is a necessity for almost all UAV swarm applications, from the military to package delivery [14]. Apart from appropriately handling user and asset telemetry and data information, network agents need to be able to discern which information and other agents they can trust. Similarly, agent failures must be isolated and detected to avoid disruption of the entire network's desired effects. Further, techniques must be developed which would allow agents to complete their missions in hostile environments. That is, agents should possess the ability to detect attacks and vulnerabilities but also accomplish their goals even in their presence. This is especially of interest to the Department of Defense, whose assets will undoubtedly face this type of operating environment.

1.1.2 Formation Tracking with Adversarial Inputs

Formations, or organized structures and assemblies of agents within a MAS, is a widely studied topic since it is assumed to be a supporting capability of nearly all MAS applications. Indeed, formation control finds use in surveillance, search and rescue, and unknown environment exploration [15–17]. Formations described by underlying rigid graphs are shown to be stabilized by maintaining desired distances between agents [18]. It is sufficient to maintain distances relative to a few agents rather than all others. This is somewhat intuitive, since this is largely the technique behind close formation flying taught to fighter pilots. The workload of maintaining positions with respect to many other aircraft would be excessive, yet the formation can be maintained by restricting a pilot's attention to one or two lead aircraft. Because of this finding, as well as the observation that undirected formations are problematic in terms of stabilization [19], directed formations have received growing research attention. Separately, directed formations enjoy certain practical advantages like reduced sensing, communication, and processing capabilities [20]. We consider the formation tracking problem, which entails agents assembling into a desired formation which moves together at a common velocity. The goal is to design a distributed control law relying only on local measurements that drives all agents to the specified formation shape and the same constant velocity. [21] accomplishes this goal using the concept of target points and integral control. The target points are defined based on the relative position of an agent to its leaders, or the other agents it is charged with maintaining distances from. Integral control drives all agents to their target points, which in turn converge to the same velocity. As these points are achieved, so is the goal.

Hostile environments increase the risk of unwanted intrusions in the operation of such activities. Taking this into consideration, a many attacks can be reduced to an adversarial control input acting on the dynamics of the system. Likely, these will occur at the agent level. For instance, one UAV in a formation could be subject to a cyber-physical attack producing improper motor control. To achieve the formation tracking goal, it is critical that the agent can compensate with proper inputs in order to prevent other agents from reaching their target points. The control formulated in [21] fails to achieve the goal in the presence of such inputs. Therefore, methods of attenuating adversarial inputs are desired to be incorporated. However, the actual effect an adversary can have on the system varies widely. Adaptive machine learning methods are promising since they have the possibility of accommodating for varying and changing attacks. Motivated by the desire to compensate for such unknown adversary behavior, the problem is formulated as a two-player zero-sum game, similar to the problem considered in H_∞ control [22]. Here, the operator and the adversary have opposite objectives. A zero-sum game implies that the modeled adversary will exhibit the worst-case behavior, which is useful in developing generalized techniques. However, H_∞ control methods are highly technical and usually require solutions to very complex nonlinear partial differential equations. Policy iteration is a useful method for approximating solutions to these games, but this process is most easily done offline. In the case of multi-agent formation control, online solutions are required to allow adaptation to changing environments and threats. The authors in [23] formulate an online solution that is limited by an excitation requirement. Finally, the research in [24] presents an online adaptive solutions which uses memory replay to avoid the excitation requirement based on the research in [25]. This forms the basis of the method we use for adversary attenuation in the formation tracking problem.

1.1.3 Attack Detection

A critical ability for secure autonomous systems is to recognize when they are subject to some sort of attack, whether physical, cyber-physical, or strictly cyber in nature. In their systemization of knowledge on UAV security and threats, Nassi et al. identifies detection of attacks as a highly-relevant and under-served problem [26]. The ability to reliably detect incoming/on-going attacks to their vehicles would provide operators with the situational awareness to take corrective measures. In many cases, simply being aware of the threats would allow operators to mitigate effects by initiating safe modes or aborting sensitive aspects of the mission. Especially in military applications, it is critical that operators obtain this capability, since a security breach could give adversaries access to sensitive information or even control over the aircraft. Most attack detection schemes can be grouped into three main categories: signature-based, unsupervised learning, and supervised learning.

Signature-based attack detection methods are often the most effective for the certain attacks they were designed for. Their inherent flaw is their limited generality and scope. For example, [27] and [28] propose nice methods of detecting GPS spoofing attacks, which are arguably the most prominent ones. However, these solutions provide no zero-day protection and were formulated after these attack rose in prevalence. Additionally, they only detect the GPS spoofing attacks in accordance with their model, giving no room for how the attack methods may shift as adversaries become ever more creative. Similarly, the software overhead for implementing these detection schemes is seemingly without upper limit, since each purpose-built program would need to be included for comprehensive detection systems. The lack of generality requires operators and developers to have knowledge of many different fields rather than simply aircraft operation and production. In [29], the authors consider the same set of cyber-physical attacks presented here and use a control invariant approach for detection. Their software learns the vehicle's operational signature and recognizes abnormalities corresponding to attacks. It is a very promising solution, but the field would still benefit from a more generalized model which requires less technical information about the vehicle. Other prominent attacks exploit data vulnerabilities in transmission and storage such as deletion and deauthentication [30]. Traditional cybersecurity measures like WPA are suitable to detect and protect these attack surfaces [26]. Still, it is unclear what other attack methods adversaries will adopt, and these traditional protection measures lack the ability to identify abnormal activity other than that which corresponds to their design signature.

In order to automate the attack detection process and increase the zero-day attack protection, security researchers have turned to machine learning methods. These schemes usually learn normalities during training and issue a warning based on deviation from these. Unsupervised learning models require no labeled ground truth data for training but are usually prone to considerably high false positive rates [26, 29, 31]. On the other hand, supervised learning models require labeled training data, which can be hard to acquire for some applications. Still, especially when simulation results can be used, producing labeled training data can be well-worth the effort. Supervised methods usually outperform unsupervised ones and have shown promise in related fields like intrusion detection in smart infrastructure [31] and detection of drone presence by network traffic analysis and acoustics [32, 33]. Machine learning models are particularly convenient because they do not require a priori construction of a formal system model. They automatically recognize trends and connections in the data they provided. For attack detection in UAVs, a behavior-based model is highly desired since it would utilize data already available to the flight computer such as telemetry and state measurements. This means no extra hardware or data collection is required, which is ideal to keep the detection scheme's footprint on the overall system as small as possible.

Although gaining the situational awareness of successfully detecting attacks is prized on its own, detection begs the question of further mitigation. Once detected, mitigation schemes have been proposed such as back-tracking commands or automatically initiated safe modes [26, 29]. It is proposed that a mechanism which stores commands might be able to reverse-execute them to lead the compromised vehicle back home. In the worst case, multiple companies [34, 35] count on the reliability of a deployable safety parachute which could be used if all else fails, although this would require physical recovery of the landed system.

1.1.4 Seizure Prediction

The Epilepsy Foundation of America states that 1 in 26 Americans will develop epilepsy in their lifetime [36]. A neurological disorder characterized by chronic, recurrent, and unprovoked seizures, the condition can have a debilitating effect on the afflicted person's life. During a seizure, the patient is at a greatly increased risk of injury and even death [37]. Injuries are often a result of the sudden seizure onset, as patients and caretakers have no indication that one is imminent and are unable to assume safer positions. In some cases, death is even attributed to lack of warning, as the onset of seizure caused a patient to enter a dangerous but preventable situation like drowning or falling [38]. These cases fall under the umbrella of Sudden Unexpected Death in Epilepsy, or SUDEP. SUDEP is a significant risk for those living with epilepsy, and associated seizures may even cause suffocation from severe apnea or heart failure. The cumulative effect of the prevalence of the condition and its associated risk factors is a severely detrimented quality of life for the patient. While some medications and seizure-suppression therapies do exist, a patient's quality of life would be greatly improved with simple warning of an oncoming seizure. Much of the associated risk can be minimized by assuming a safe position/environment and alerting an observer.

Motivated by the potential to improve the lives of so many people, research interest from scientists and medical professionals has increased in the past decade [39]. Because seizures are a disruption of the electrical communication between neurons in the brain [36], the research is focused on analysis of electroencephalogram (EEG) signals. Some patients report an indescribable "sense" before seizure onset, which indicates preictal (just prior to seizure onset) brain activity may be detectable [40]. Many different features have been proposed which attempt to distinguish between preictal and interictal (normal) brain activity with varying degrees of success. Such features include the largest Lyapunov exponent, correlation density, dynamical similarity index, etc. [39]. Recently, however, advances in artificial intelligence and deep learning have enabled novel, automated feature extraction and classification methods. The studies in [41] and [42] present state-of-the-art performance using deep learning convolutional neural networks (CNNs) for feature extraction and classification. The same studies also suggest that the distinguishing features between preictal and interictal activity can be different between patients and even in the same patients over time. Therefore, generalized methods which account for this are highly desired for prolonged effectiveness. Here, CNN machine learning models have a distinct advantage, since the distinguishing features are taken directly from the training data itself, independent of time and patient.

Our involvement in this problem is the result of a collaboration request from neuroscience researchers at Yale University. This research group's experience with applications of machine learning led them to seek assistance with seizure prediction analysis of EEG signals taken in mice for testing of an epilepsy therapy in development. The initial provided mice data is much less rich than available human data because of the smaller brain size and number of available electrodes. As such, we demonstrate the analysis in a dataset gathered from human patients to better compare it to existing methods. We sought this as an opportunity to apply a hybrid method of time signal analysis developed in another part of this work to a different field to increase its scope and demonstrate it's effectiveness with other data. This could be extending to data which is further applicable in MAS and aerospace systems in general.

1.1.5 Blockchain

Blockchain has been suggested as a potential solution for the need of a secure communication and coordination pathway for UAV networks. Blockchain technology originated in 2008 when Satoshi Nakamoto applied it to his cryptocurrency in the white paper *Bitcoin: A Peer-to-Peer Electronic Cash System* [43]. Bitcoin is a decentralized, peer-to-peer digital currency that retains value with no support from a central authority. The blockchain is at the heart of the cryptocurrency's ability to exchange economic value in a distributed manner. The blockchain is a public, chronological ledger of transactions recorded by agents on the network. The ledger is kept and updated by every agent in the network. Transactions are grouped into datasets that make up blocks. Certain information is included in each block, such as specific transactions, a reference to the previous block to keep chronological order, and the answer to a computationally difficult problem known as "proof of work". Inclusion of "proof of work" data validates each individual block and makes the creation of overall blocks computationally difficult in order to prevent malicious agents from illegitimately manipulating the blockchain [43]. Proof of work is sometimes referred to as the block's digital fingerprint. It is based on cryptographic techniques that create unpredictable outputs for different inputs [5]. Therefore, if a block is manipulated, the digital fingerprint will be completely different, allowing detection. After verifying all transactions on a block do not conflict with transactions from other blocks, the participating agent, known as a miner, then adds the block to the end of the blockchain. Once this happens, the information in the block cannot be destroyed or changed, and it is made public to the entire network. The blockchain is stored and updated periodically by every agent in the network in a local, peer-to-peer fashion. Overall, the blockchain is a mechanism for a distributed network to keep an accurate and secure record without the need for a central authority.

While a blockchain does not require a central authority, it is considered global information since all agents operate on it. It might seem counterintuitive to integrate global information into UAV swarms since their value lies in the decentralization associated with local interactions. However, it has been shown that in some instances [44, 45] global information in UAV swarms can be beneficial. This suggests a combination of the two types of information might be optimal for certain applications without sacrificing robustness and scalability [46]. Additionally, recent advances in processing and communications hardware allow large scale communication and global knowledge dissemination in UAV networks that might not have been technologically capable otherwise, as [5] points out. Research has identified the need for agents to detect and trust each other [47]. Likewise, it has been demonstrated that malicious or even faulty agents are able to disrupt the entire swarm from achieving its goals [48]. Digital security, separate from only robot swarms, hinges on "core services such as data confidentiality, data integrity, entity authentication, and data origin authentication" [5]. [49] shows that a lack of practical solutions to security problems exists for UAV swarms compared to many other fields. It is possible this is due to the academic community being preoccupied with the more fundamental swarm problems like dynamics, control, autonomy, and interaction. The community suggests that addition of a blockchain into swarms is inherently more secure since a common ledger is kept by all agents of the network, rather than in one or a number of vulnerable points of failure [5, 9, 14, 47, 47, 49–54]. Additionally, it provides a framework where general cryptographic methods, rather than solutions specific to UAV networks, could be used for information security in drone swarms. [5] asserts that blockchain could provide a communications channel that is both peer-to-peer and secure, and [51] agrees by listing a host of cybersecurity threats to UAV systems that a blockchain would be resilient against. [5] offers public key and digital signature cryptography on blockchain as a possible solution for both transmitting secure information to the correct receiver only and authenticating the original agent's identity for information that is public. In both cases, data is exchanged securely over a common blockchain regardless of the presence of malicious or faulty agents. [14] shows that blockchain technology enables UAV networks to detect and avoid corroborating faulty information from hijacked agents. The security issue is made even more complicated when inter-service cooperation is required between agents of networks operated by multiple vendors. [50] showed that a blockchain-based solutions exist for this problem, which will be especially prevalent in areas of very dense swarm operations. More research is needed in the form of simulations and experiments that can validate this method as a means of achieving more secure and robust MAS.

1.1.6 Conclusion

Overall, there are many open questions in the field of counter autonomy for aerial multi-agent systems. Control and operations of autonomous MAS benefits from a very healthy stream of research interest. Because of this, the field has greatly matured, and it now seems as though ensuring these control techniques can be implemented safely and reliably will emerge as a major obstacle. This work attempts to build on the progress of the others listed and develop techniques of ensuring aerial systems can securely fulfill the promises their technologies offer. As noted, we seem to be on the verge of a great boom in the adoption of autonomous systems in both the military and commercial realms, and one of the main challenges plaguing the field is defense against such broad and unpredictable threats. Therefore, it seems fitting to investigate techniques of resilience in these systems. Advances in computation and artificial intelligence uniquely position themselves as ideal options for consideration in these problems. Hopefully, the methods developed here can also be used to improve the quality of life for epileptic patients as well.

1.2 Contributions

The main contributions of this work are

- A method of achieving distributed formation trajectory tracking control while attenuating adversarial control inputs using machine learning
- A method and proof of concept of behavior-based cyber-physical attack detection in autonomous aircraft using supervised and on-board learning
- A hybrid method of time signal analysis using both convolutional neural networks and support vector machines which is demonstrated in epileptic seizure prediction
- Investigation of a blockchain method of managing Byzantine agents in a distributed network

2. FORMATION TRACKING IN THE PRESENCE OF ADVERSARIAL INPUTS

2.1 Introduction

Formation control for teams of autonomous agents has found extensive applications in exploration, surveillance, and industrial operations. One essential ability of formations is tracking control, so that the formation may properly follow the leader and specified goal. In the spirit of redundancy, accomplishing this in a distributed manner is desirable to ensure mission completion and reduce the reliance on global information spread. Here, distributed control of the formation involves the formulation of agent control laws which rely only on local measurements. The desired formation can be described by a directed graph which specifies the distance constraints between agents. By maintaining the prescribed distances, the formation can be stabilized. In a directed formation, we say that agent j is a *leader* of agent i if agent i must maintain a prescribed distance from agent j. Likewise, agent i is considered a *follower* of agent j.

In real-time applications, it is likely that formations of agents may be operating in hostile environments and exposed to a variety of adversaries such as cyber-physical attacks, network attacks, or even adverse weather conditions. Here, the goal is formulate distributed control laws for each agent such that the formation shape is maintained while all agents converge to move at the same velocity, even when agents are subject to erroneous adversarial inputs. Formations described by underlying rigid graphs are shown to be stabilized by maintaining desired distances between agents [18]. We follow the method of [21] and consider acyclic, minimally rigid formations and utilize the idea of tracking target points for each of the agents. We consider adversarial inputs for each of the agents in the form of a zero-sum game such that the worst-case adversary is accommodated, as in [24]. Thus, the overarching problem involves formation flight, velocity consensus, and adversary attenuation.

2.2 Problem Formulation

We consider the class of acyclic, minimally rigid formations in the plane $\mathbb{R}^2$ with $n \geq 2$ agents. In [55], it is shown that any n-agent formation can be constructed from Henneberg vertex addition starting from a two agent formation such that for agents i where $\{i \mid 3 \leq i \leq n\}$, two other agents $j, k \in \{1, 2, ..., i-1\}$ are selected as its leaders. Agent 1 is usually referred to as the *global leader* of the formation, and agent 2 is often called the *first follower*, since it only maintains a single distance. Simply, the first follower maintains a prescribed distance from the global leader, while all other followers are added to the formation via vertex addition and maintain prescribed distances from two preceding agents. This class of formations is referred to as *vertex-addition formations*. Examples of these formations are shown in Figure 2.1, where 2.1(e) is not a vertex-addition formation since agents 3 and 4 each only have one leader (and the graph is cyclic). For agents with two leaders, the orientation of the follower to its two leaders can be described as either clockwise or counter-clockwise. Starting from the follower, the successive direction of its two leaders determine its orientation. Figure 2.1(a) and 2.1(b) give examples of this.

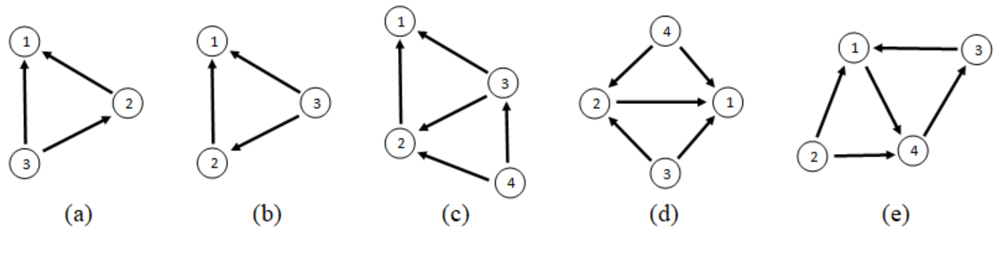


Figure 2.1. Vertex-addition formations. (a) is oriented clockwisely, (b) is oriented counter-clockwisely, and (e) is not a vertex-addition formation.

Each agent's motion is described by the simple point kinematic model

$$\dot{x}_i = u_i, \qquad \forall\, i = 1, 2, ...n \tag{2.1}$$

where $x_i \in \mathbb{R}^2$ is the position of agent i in the plane and u_i is agent i's control input.

The distributed formation tracking problem assumes the global leader's velocity converges exponentially fast to a constant, v_0. Each agent i has a local measurement of its relative position to its leader j, $x_i - x_j$. As such, u_i must be chosen for $i = 2, 3, ..., n$ in terms of its measurements so the formation converges to the desired shape and all agents reach the velocity consensus v_0. Following the target point method in [21], the formation can be maintained by defining target points for each agent to track based on the relative position of its leaders. In a vertex-addition formation, agent 2 has a single leader to follow while agents $i > 2$ have two leaders to follow. As such, there are separate target point formulations for each case.

2.3 Existing Formation Control

2.3.1 One-leader Target Point

Considering a directed two-agent formation, let $x, y \in \mathbb{R}^2$ denote the positions of the follower and the leader, respectively. If $\dot{y} = v$ where $v \rightarrow v_0$ exponentially fast and follower x knows its relative position measurement, $x - y$, the leaders velocity direction, $r = \dfrac{v}{||v||}$ ($r = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ if $v = 0$), and the prescribed distance away from the leader, d. Then, a *target point*, $\tau(y)$ can be defined as

$$\tau(y) = y - dr \tag{2.2}$$

Using this definition, $||\tau(y) - y|| = d$ and $\tau(y)$ moves at the same velocity as the leader. The target point represents the desired position of the follower and, as such, the tracking goal for the first follower.

2.3.2 Two-leader Target Point

Considering a directed three-agent formation, let $x, y_1, y_2 \in \mathbb{R}^2$ denote the positions of the follower, the first leader, and the second leader, respectively. The leaders' dynamics are given as

$$\dot{y}_1 = v_1, \qquad \dot{y}_2 = v_2 \tag{2.3}$$

and d_1 and d_2 are the prescribed following distances from the leaders. We can assume that by the chosen leader selection $d_1 \geq d_2$ and $v_1, v_2 \to v_0$ exponentially fast. Under these conditions, there are four possible cases which depend on the following distances and the leader positions. Figure 2.2 illustrates these with the black square representing the target point.

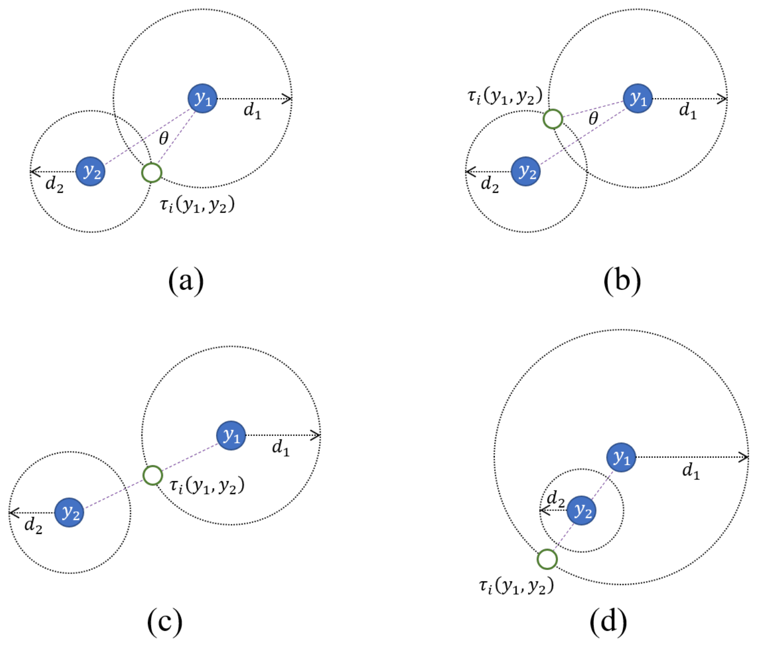


Figure 2.2. Representation of the four cases for two-leader target points from [21]. (a) $d_1 - d_2 < ||y_1 - y_2|| < d_1 + d_2$ and the formation is oriented counter-clockwisely, (b) $d_1 - d_2 < ||y_1 - y_2|| < d_1 + d_2$ and the formation is oriented clockwisely, (c) $||y_1 - y_2|| \geq d_1 + d_2$, and (d) $0 < ||y_1 - y_2|| \leq d_1 - d_2$.

In Figure 2.2(a) and (b), the distances are large enough such that the triangle inequality holds and the target point is then wholly defined by the formation orientation. In Figure 2.2(c) and (d), the prescribed distances do not allow simultaneous fulfillment. Therefore, we define the desired goal as maintaining the proper distance to the first follower while also being as close as possible to the second leader. From these four cases, we can define

$$s = \begin{cases} 1 + \dfrac{[||y_1 - y_2|| - (d_1 + d_2)]\,[||y_1 - y_2|| - (d_1 - d_2)]}{2d_1||y_1 - y_2||}, & d_1 - d_2 < ||y_1 - y_2|| < d_1 + d_2 \\ 1, & \text{otherwise} \end{cases} \tag{2.4}$$

where s represents $cos(\theta)$ for $d_1 - d_2 < ||y_1 - y_2|| < d_1 + d_2$. Then a rotation matrix through the angle θ can be defined as

$$R_\tau(s) = \begin{bmatrix} s & \gamma_f\sqrt{1-s^2} \\ -\gamma_f\sqrt{1-s^2} & s \end{bmatrix} \tag{2.5}$$

where

$$\gamma_f = \begin{cases} 1, & \text{if the formation is clockwisely oriented} \\ -1, & \text{otherwise} \end{cases} \tag{2.6}$$

so that $R_\tau(s)$ rotates a vector in accordance with the formation orientation. Then, a target point, $\tau(y_1, y_2)$ can be defined as

$$\tau(y_1, y_2) = \begin{cases} y_1 + \dfrac{d_1}{||y_1 - y_2||} R_\tau(s)(y_2 - y_1), & y_1 \neq y_2 \\ y_1, & \text{otherwise} \end{cases} \tag{2.7}$$

Using this definition, when $d_1 - d_2 < ||y_1 - y_2|| < d_1 + d_2$, $||\tau(y_1, y_2) - y_1|| = d_1$ and $||\tau(y_1, y_2) - y_2|| = d_2$. In the other two cases, $||\tau(y_1, y_2) - y_1|| = d_1$ while $||\tau(y_1, y_2) - y_2||$ is minimized. Therefore, when an agent occupies its target point, it is in the proper position described by the formation. The target point represents the desired position of the follower and, as such, the tracking goal for each agent.

2.3.3 Distributed Formation Tracking Control Policies

In [21], the integral control based on target points is presented which accomplishes the distributed formation tracking problem without adversarial inputs. Their main result yields the desired control policy

$$\begin{aligned} u_{d_i} &= -\left(x_i - \tau_i\right) + w_i \\ \dot{w}_i &= -\left(x_i - \tau_i\right) \end{aligned} \tag{2.8}$$

where $x_i \in \mathbb{R}^2$ denotes agent i's position in the plane and τ_i represents its calculated target point. Note that since the target points only require locally available information, the listed control accomplishes the formation tracking goal in a distributed manner. We refer to [21] for proof of convergence for both target point cases and will use this policy as the desired control for each agent u_{d_i}. By *desired control*, we mean a control policy which will achieve the formation tracking goal nominally, or without consideration of adversaries.

2.4 Adversarial Inputs

The target points defined in sections 2.3.1 and 2.3.2 define the goal for distributed formation tracking. However, we are further concerned with the problem of distributed formation tracking in the presence of adversarial inputs. As such, u_i from the kinematic model given in equation 2.1 is given as

$$u_i = u_{c_i} + k_i(x_i)u_{a_i}, \qquad \forall\, i = 1, 2, ...n \tag{2.9}$$

where u_{c_i} is the control input including adversary compensation that we seek to formulate and the additional term describes the general nonlinear adversary. If there is a desired trajectory expressed by x_{d_i} with dynamics given by $\dot{x}_{d_i} = u_{d_i}$ which will accomplish the distributed formation tracking goal ignoring adversarial inputs, we can define a tracking error due to the adversary as

$$e_i = x_i - x_{d_i} \tag{2.10}$$

with dynamics

$$\dot{e}_i = u_i - u_{d_i} \tag{2.11}$$

Letting $u_{c_i} = u_{d_i} + u_{o_i}$, the system's implemented control law, be split into the part required to achieve the desired trajectory and a part compensating for the adversarial input, equation 2.11 is rewritten as

$$\dot{e}_i = u_{o_i} + k_i(e_i)u_{a_i} \tag{2.12}$$

Note that since u_{d_i} is chosen as shown in section 2.3.3, x_{d_i} is simply the trajectory the agent would follow under the control policy from [21] without adversarial inputs. Therefore, we seek to formulate a control policy which will compensate for the adversarial inputs and drive the agent's trajectory as close as possible to x_{d_i}.

Now, following the methods in [24], we define a cost function for each agent over the tracking objective as

$$J_i\left(e_i(0), u_{o_i}, u_{a_i}\right) = \int_0^\infty Q_i(e_i) + R_i(u_{o_i}) - \gamma_i^2 \left\|u_{a_i}\right\|^2 \mathrm{d}\tau \tag{2.13}$$

where $Q_i(e_i)$ is positive semidefinite and represents the cost due to the position error, $R_i(e_i)$ is positive definite and represents the cost due to the compensating control input, and $\gamma_i \geq \gamma_i^* \geq 0$. γ_i^* is the H_∞ gain given in equation 2.14. This is a measure of adversary attenuation in the system. To attenuate the adversary to the maximum extent possible, it is desired that γ_i is as small as possible. Therefore, γ_i^* is the smallest gain that still yields a stable closed-loop system.

$$\frac{\int_0^{\mathrm{T}} \left|\left|Q_i(e) + R(u_{o_i})\right|\right|^2 dt}{\int_0^{\mathrm{T}} \left|\left|u_{a_i}\right|\right|^2 dt} \leq \gamma^2 \tag{2.14}$$

The generally nonlinear definitions of the functions $Q_i(e_i)$ and $R_i(e_i)$ provide for useful applications within this framework. For example, $R_i(e_i)$ might be used to account for input saturation constraints using a proper function which maps $e_i \in \mathbb{R}^2$ onto the interval $\left[-u_{o_{i\max}}, u_{o_{i\max}}\right]$.

Thus, our goal is to design a control strategy, u_{c_i}, to track the desired trajectory x_{d_i} as closely as possible and attenuate the effects induced by the adversary u_{a_i} while simultaneously optimizing the tracking cost function given in equation 2.13.

2.5 Adversary Compensation

This is the formulation of a zero-sum game where the adversary seeks to maximize the cost-to-go and the operator's compensating input seeks to minimize it. So, we must find the following optimal cost function for each agent

$$C_i^*(e_i(t)) = \min_{u_{o_i}} \max_{u_{a_i}} \int_t^\infty Q_i(e_i) + R_i(u_{o_i}) - \gamma_i^2 \left\| u_{a_i} \right\|^2 d\tau, \quad t \geq 0 \tag{2.15}$$

subject to the dynamical constraint given in equation 2.12.

The operator and the adversary have opposite objectives, so it is intuitive to search for a saddle point equilibrium in the zero-sum game. From game theory, this two-player optimal control problem has a unique solution if a saddle point (representing the pair of optimal control policies $u_{o_i}^*$ and $u_{a_i}^*$) exists. This saddle point is defined by the Nash condition

$$\min_{u_{o_i}} \max_{u_{a_i}} J_i\left(e_i(0), u_{o_i}, u_{a_i}\right) = \max_{u_{a_i}} \min_{u_{o_i}} J_i\left(e_i(0), u_{o_i}, u_{a_i}\right). \tag{2.16}$$

This situation represents the worst-case adversary, since from this equilibrium condition, a unilateral change by either player results in decreased performance for the players. That is,

$$J_i\left(e_i(0), u_{o_i}^*, u_{a_i}\right) \leq J_i\left(e_i(0), u_{o_i}^*, u_{a_i}^*\right) \leq J_i\left(e_i(0), u_{o_i}, u_{a_i}^*\right). \tag{2.17}$$

Figure 2.3 illustrates this concept, where the red sphere indicates the saddle point equilibrium. If this sphere were to unilaterally shift in either direction, the reward for that player would decrease in accordance with equation 2.15.

Using Leibniz's formula to differentiate the cost-to-go given in equation 2.15, the differential equivalent to the integral is in the form of the Hamiltonian, H, of the dynamics from equation 2.12 and the cost from equation 2.13, which is

$$H_i = Q_i(e_i) + R_i(u_{o_i}) - \gamma_i^2 \left\| u_{a_i} \right\|^2 + \nabla C_i^T (u_{o_i} + k_i(e_i) u_{a_i}) = 0. \tag{2.18}$$

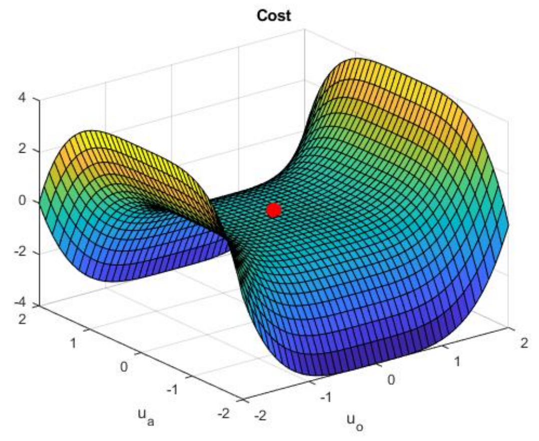


Figure 2.3. Saddle point equilibrium which arises from the Nash condition on the zero-sum game. The red sphere represents the equilibrium point at optimal control policies for both players.

where $\nabla C_i = \dfrac{\partial C_i}{\partial e_i}$. This is known as the zero-sum Bellman equation, which is a partial differential equation for the cost-to-go. Stationarity conditions on the Hamiltonian give equations for the optimal control policies

$$\frac{\partial H_i}{u_{o_i}} = 0 \Rightarrow u_{o_i}^* = -\frac{1}{2} R_i^{-1} \nabla C_i^* \tag{2.19}$$

$$\frac{\partial H_i}{u_{a_i}} = 0 \Rightarrow u_{a_i}^* = \frac{1}{2\gamma^2} k_i^T(e_i) \nabla C_i^*. \tag{2.20}$$

Substituting these optimal policies into equation 2.18 yields the Hamilton-Jacobi-Isaacs (HJI) equation

$$H_i^* = Q_i(e_i) - \frac{1}{4} \nabla C_i^{*T} R_i^{-1} \nabla C_i^* + \frac{1}{4\gamma_i^2} \nabla C_i^{*T} k_i(e_i) k_i^T(e_i) \nabla C_i^* = 0 \tag{2.21}$$

Thus, the solution of the HJI equation for C_i^* would yield the optimal operator and adversary control policies for the game. However, this equation is not generally solvable.

2.5.1 Quadratic Case

It is useful to first discuss a simple form of the problem which yields a nice analytical result and highlights the necessity of approximation methods in more general cases. The quadratic cost-to-go function is

$$C_i^*(e_i(t)) = \min_{u_{o_i}} \max_{u_{a_i}} \int_t^\infty e_i^T Q_i e_i + u_{o_i}^T R_i u_{o_i} - \gamma_i^2 \left\| u_{a_i} \right\|^2 d\tau, \quad t \geq 0 \tag{2.22}$$

where $Q_i \in \mathbb{R}^{2\times2}$ is positive semidefinite, $R_i \in \mathbb{R}^{2\times2}$ is strictly positive and symmetric, and $k_i(e_i)u_{a_i} = D_i$ with $D_i \in \mathbb{R}^{2\times1}$. When the cost-to-go function is fully quadratic, the HJI equation (2.21) reduces to the game algebraic Ricatti equation

$$0 = Q_i - P_i^T R_i^{-1} P_i + \frac{1}{\gamma_i^2} P_i D_i D_i^T P_i \tag{2.23}$$

where P_i is positive semidefinite, symmetric by design and chosen as the stabilizing solution. This equation is solvable using traditional Ricatti equation methods. This gives the optimal operator and adversary control policies

$$u_{o_i}^*(t) = -R_i^{-1} P_i e_i(t) \tag{2.24}$$

$$u_{a_i}^*(t) = \frac{1}{\gamma^2} D^T P_i e_i(t). \tag{2.25}$$

However, this solution only holds for the quadratic case. Still, it is a useful comparison for more general methods, and is worth highlighting here. In the non-quadratic case, it is generally more efficient or even distinctly necessary to approximate and iterate towards the solution to equation 2.21.

2.5.2 Online Solution by Approximation via Neural Networks

Recent research on methods of approximating solutions of HJI equations using adaptive deep learning methods has shown promise and is the motivation behind applying these methods to formation control here. We follow the methods of [24] and will use actor-critic reinforcement learning by policy iteration. Figure 2.4 illustrates this process, which successively iterates toward the solution to equation 2.21. Once this solution is converged upon, ∇C_i is known and both actors will be implementing the optimal policies in accordance with equations 2.19 and 2.20. The critic evaluates

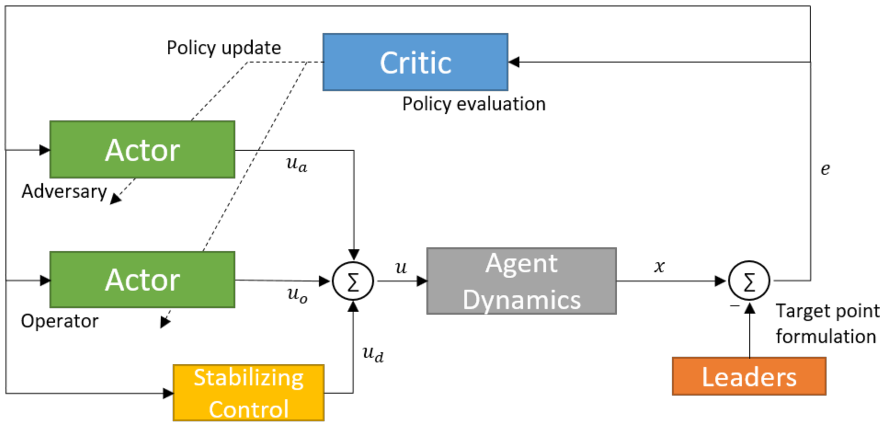


Figure 2.4. Two-player actor-critic reinforcement learning by policy iteration. Implemented policies are functions of the critic evaluation. Learning is finished when critic evaluations remain constant to a specified degree and the optimal actor control policies are implemented.

control policies implemented by the actors and approximates a solution to the HJI equation. The two actors implement optimal control policies for the given policy evaluation. Once the policy evaluations converge to the saddle point defined by equation 2.16, the final optimal control policies will be in place.

Perhaps the simplest and most intuitive way to iterate towards a solution using this method is to successively iterate control policies for both actors based on policy evaluation and check for convergence. Upon convergence, implement the resulting control policies. However, this method is not practical for use in real systems since the learning is done by iteration offline. A better solution is a method of converging toward this solution online while operating adaptively. This requires simultaneous update of actor policies and critic evaluation such that all three are coupled. One way of achieving this is directly approximating the solution to the HJI equation using neural networks, as in [24]. Using this method, the critic and both actors are approximated using separate neural networks. The key idea behind this is function approximation using neural networks, which is shown in [56] to have bounded approximation error for a fixed number of neurons or vanishing approximation error as the number of neurons goes to infinity in accordance with the Weierstrass higher order function approximation theorem. Then, the problem reduces to implementing real-time coupled weight tuning laws such that the weights converge to their optimal value, which is the smallest possible approximation error for a fixed number of neurons. Therefore, we must define and consider each approximator.

2.5.3 Critic

The critic approximator estimation can be written as

$$\hat{C}_i(e_i) = \hat{W}_{c_i}^T \Phi_{c_i}(e_i) \tag{2.26}$$

where $\hat{W}_{c_i}$ represents the $N \times 1$ vector of neural network weights, $\Phi_{c_i}(e_i)$ represents the $N \times 1$ vector of activation functions for the critic approximator, and N is the number of basis functions (number of neurons). Thus, the ideal critic approximator for a fixed N is

$$C_i^*(e_i) = W_{c_i}^{*T} \Phi_{c_i}(e_i) + \epsilon_{c_i}(e_i) \tag{2.27}$$

where $W_{c_i}^*$ are the ideal weights and $\epsilon_{c_i}(e_i)$ denotes the online approximation error. The ideal critic approximator has the derivative

$$\frac{\partial C_i^*}{\partial e} = \nabla \Phi_{c_i}^T W_{c_i}^* + \nabla \epsilon_{c_i}. \tag{2.28}$$

where $\nabla \Phi_{c_i} = \dfrac{\partial \Phi_{c_i}(e)}{\partial e}$. Substituting this derivative into equation 2.18, gives an approximated Hamiltonian

$$H_i^* = Q_i(e_i) + R_i(u_{o_i}) - \gamma_i^2 \left\| u_{a_i} \right\|^2 + W_{c_i}^{*T} \nabla \Phi_{c_i}(u_{o_i} + u_{a_i}) = \epsilon_H \tag{2.29}$$

where

$$\epsilon_H = -\nabla \epsilon_{c_i}^T (u_{o_i} + u_{a_i}) \tag{2.30}$$

represents the residual error due to function approximation. For fixed actor control policies, the current online approximate Hamiltonian is then

$$\hat{H}_i = Q_i(e_i) + R_i(u_{o_i}) - \gamma_i^2 \left\| u_{a_i} \right\|^2 + \hat{W}_{c_i}^T \mu(t) \tag{2.31}$$

where $\mu(t) = \nabla \Phi_{c_i}(u_{o_i} + k(e_i)u_{a_i})$. Therefore, we must formulate an update law that ensures $\hat{W}_{c_i} \to W_{c_i}^*$ and $\hat{H}_i \to H_i^*$. Following model reference adaptive control from [57], we can use a deep learning approach for using past data together with current data. We define an error for the current data

$$e_{H_i} = \hat{H}_i - H_i^* \tag{2.32}$$

and for the past data

$$e_{H_{iW_k}} = \hat{H}_{iW_k} - H_i^* \tag{2.33}$$

where W_k denotes the weights at time k. The weight performance index, E_i in equation 2.34, is based on mean squared error for both current and past data, and it can be used to ensure the neural network weights are tuned in the proper direction.

$$E_i = \frac{1}{2} \frac{e_{H_i}^T e_{H_i}}{\left(\mu(t)^T \mu(t) + 1\right)^2} + \frac{1}{2} \sum_{k=1}^{k_{max}} \frac{e_{H_{iW_k}}^T e_{H_{iW_k}}}{\left(\mu\left(t_k\right)^T \mu\left(t_k\right) + 1\right)^2} \tag{2.34}$$

where k_{max} defines the size of the window of stored data. It has been shown that too much past data adversely affects performance of the system, while too little stored data has the same effect, but the analysis in [24] requires at least N linearly independent data points are stored for proper weight convergence. Thus, k_{max} is a design parameter which should be chosen based on desired performance. Finally, the critic neural network weights can be updated via gradient descent on E_i from equation 2.34. This yields the update law

$$\begin{aligned}\dot{\hat{W}}_{c_i} &= -\alpha_c \frac{\partial E_i}{\partial \hat{W}_{c_i}} \\ &= -\alpha_c \frac{\mu_i(t)}{(\mu_i(t)^T \mu_i(t) + 1)^2} \left(\mu_i(t)^T \hat{W}_{c_i}(t) + R_i(u_{o_i}(t)) + Q_i(e_i(t)) - \gamma_i^2 \left\| u_{a_i}(t) \right\|^2 \right) \\ &\quad - \alpha_c \sum_{k=1}^{k_{max}} \frac{\mu_i(t_k)}{(\mu_i(t_k)^T \mu_i(t_k) + 1)^2} \\ &\quad \left(\mu_i(t_k)^T \hat{W}_{c_i}(t) + R_i(u_{o_i}(t_k)) + Q_i(e_i(t_k)) - \gamma_i^2 \left\| u_{a_i}(t_k) \right\|^2 \right)\end{aligned} \tag{2.35}$$

where $\alpha_c > 0$ is the learning rate for the critic. We refer to [24] for the proof of convergence for $\hat{W}_{c_i} \to W^*_{c_i}$.

2.5.4 Operator Actor

Similarly, we can estimate the proper operator compensating control input with another neural network as

$$\hat{u}_{o_i}(e_i) = \hat{W}^{*T}_{uo_i} \Phi_{uo_i}(e_i) \tag{2.36}$$

where W_{uo_i} represents the $N_2 \times 2$ vector of neural network weights, $\Phi_{uo_i}(e_i)$ represents the $N_2 \times 1$ vector of activation functions for the operator actor approximator, and N_2 is the number of basis functions (number of neurons). Thus, the ideal operator actor approximator for a fixed number of neurons is

$$u^*_{o_i}(e_i) = W^{*T}_{uo_i} \Phi_{uo_i}(e_i) + \epsilon_{uo_i}(e_i) \tag{2.37}$$

where $W^*_{uo_i}$ are the ideal neural network weights and $\epsilon_{uo_i}(e_i)$ denotes the online function approximation error.

From equations 2.19 and 2.26, the optimal operator policy for any critic evaluation is

$$u^*_{o_i}(e_i) = -\frac{1}{2} R_i^{-1} \left(\nabla \Phi^T_{c_i} \hat{W}_{c_i} \right). \tag{2.38}$$

So, we can define an approximation performance index for $\hat{u}_{o_i}$ as

$$e_{u_{o_i}} = \hat{u}_{o_i} - u^*_{o_i}. \tag{2.39}$$

We desire to select weights which minimize the squared error

$$E_{u_{o_i}} = \frac{1}{2} e^T_{u_{o_i}} e_{u_{o_i}}, \tag{2.40}$$

so we can update the weights by gradient descent using

$$\dot{\hat{W}}_{uo_i} = -\alpha_{uo} \frac{\partial E_{u_{o_i}}}{\partial \hat{W}_{uo_i}} = -\alpha_{uo} \Phi_{uo_i} \left(\hat{W}^{*T}_{uo_i} \Phi_{uo_i}(e_i) + \frac{1}{2} R_i^{-1} \left(\Phi^T_{c_i} \hat{W}_{c_i} \right) \right)^T \tag{2.41}$$

where $\alpha_{uo} > 0$ is the learning rate for the operator. We refer to [24] for the proof of convergence of $\hat{W}_{uo_i} \rightarrow W^*_{uo_i}$.

2.5.5 Adversary Actor

Likewise, we can estimate the worst-case adversary control policy with a separate neural network as

$$\hat{u}_{a_i}(e_i) = \hat{W}^{*T}_{ua_i} \Phi_{ua_i}(e_i) \tag{2.42}$$

where W_{ua_i} represents the $N_2 \times 2$ vector of neural network weights, $\Phi_{ua_i}(e_i)$ represents the $N_2 \times 1$ vector of activation functions for the adversary actor approximator, and N_2 is the number of basis functions (number of neurons). Thus, the ideal adversary actor approximator for a fixed number of neurons is

$$u^*_{a_i}(e_i) = W^{*T}_{ua_i} \Phi_{ua_i}(e_i) + \epsilon_{ua_i}(e_i) \tag{2.43}$$

where $W^*_{ua_i}$ are the ideal neural network weights and $\epsilon_{ua_i}(e_i)$ denotes the online function approximation error.

From equations 2.20 and 2.26, the optimal adversary policy for any critic evaluation is

$$u^*_{a_i}(e_i) = \frac{1}{2\gamma^2} k_i^T(e_i) \left(\nabla \Phi^T_{c_i} \hat{W}_{c_i} \right). \tag{2.44}$$

So, we can define an approximation performance index for $\hat{u}_{a_i}$ as

$$e_{u_{a_i}} = \hat{u}_{a_i} - u^*_{a_i}. \tag{2.45}$$

We desire to select weights which minimize the squared error

$$E_{u_{a_i}} = \frac{1}{2} e^T_{u_{a_i}} e_{u_{a_i}}, \tag{2.46}$$

so we can update the weights by gradient descent using

$$\dot{\hat{W}}_{ua_i} = -\alpha_{ua} \frac{\partial E_{u_{a_i}}}{\partial \hat{W}_{ua_i}} = -\alpha_{ua} \Phi_{ua_i} \left(\hat{W}^{*T}_{ua_i} \Phi_{ua_i}(e_i) - \frac{1}{2\gamma^2} k_i^T(e_i) \left(\nabla \Phi^T_{c_i} \hat{W}_{c_i} \right) \right)^T \tag{2.47}$$

where $\alpha_{ua} > 0$ is the learning rate for the adversary approximator. We refer to [24] for the proof of convergence of $\hat{W}_{ua_i} \to W^*_{ua_i}$.

2.6 Distributed Formation Tracking with Adversarial Inputs

Finally, we can now synthesize the results from the above sections to create a method of achieving distributed formation tracking even when agents are subject to adversarial inputs. The weight tuning equations (2.35, 2.41, and 2.47) allow for simultaneous weight updates every time an error measurement is taken. The process is defined in Algorithm 1.

Algorithm 1: Deep Learning Formation Trajectory Tracking under Adversarial Inputs using Target Points

```
Result: Distributed formation trajectory tracking which attenuates
        adversarial inputs
Establish formation leader agent $y$ and ensure $\dot{y} \to v_0$
Start with initial positions of all agents $x_i(0) = x_{d_i}(0)$ and $e_i(0) = 0$
foreach follower agent $i$ do
    /* construct goal formation and start with given neural
       network weights                                              */
    Define follower distances, $d_1$ when $i = 1$ or $d_{1_i}, d_{2_i}$ when $i > 1$, s.t. the
      desired formation is constructed using vertex addition
    Initialize neural network weights, $\hat{W}_{c_i}(0), \hat{W}_{uo_i}(0), \hat{W}_{ua_i}(0)$ randomly or
      otherwise
end
for $m = 1, 2, ...$ do
    /* each time $e_i(t)$ is measured where $m$ is the index of
       measurement                                                  */
    foreach follower agent $i$ do
        Propagate $t_i$ and $e_i(t)$ forward using equations 2.12, 2.36, and 2.42
        Propagate $\hat{W}_{c_i}, \hat{W}_{uo_i}, \hat{W}_{ua_i}$ using equations 2.35, 2.41, and 2.47
        Compute $\hat{C}_i(e_i) = \hat{W}_{c_i}^T \Phi_{c_i}(e_i)$, (policy evaluation)
        Compute $\hat{u}_{o_i}(e_i) = \hat{W}_{uo_i}^{*T} \Phi_{uo_i}(e_i)$, (operator policy)
        Compute $\hat{u}_{a_i}(e_i) = \hat{W}_{ua_i}^{*T} \Phi_{ua_i}(e_i)$, (worst-case adversary policy)
        Compute $\tau_i$ as in equation 2.2 $(i = 1)$ or 2.7 $(i > 1)$
        Compute $u_{d_i}$ as in equation 2.8
        Implement control policy $u_{c_i} = u_{d_i} + \hat{u}_{o_i}$
        if $m < k_{max}$ then
            /* if history stack is not full, add to it              */
            Record $\mu_i(t_m)$, $e_i(t_m)$, $\hat{u}_{o_i}(t_m)$, $\hat{u}_{a_i}(t_m)$ in history stack
        end
    end
end
```

Note that Algorithm 1 defines the process of creating a formation which will perform distributed trajectory tracking while attenuating adversarial inputs at each agent.

2.7 Simulation Results

Here, simulation results are provided in order to demonstrate the effectiveness of Algorithm 1. In the first scenario, a simple three agent formation is described. When subject to persistent sinusoidal attacks, $u_a = \begin{bmatrix} \cos t + \sin 3t \\ \cos 3t + \sin t \end{bmatrix}$, the control law defined in [21] fails to achieve both formation assembly and velocity consensus. Figure 2.5 shows the distance error for each edge which describes the desired formation.

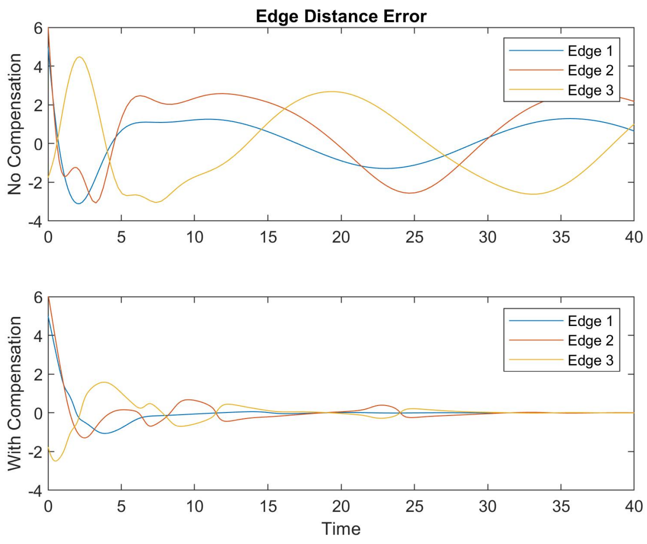


Figure 2.5. Edge distance error over time for both cases. Notice that with compensation the formation is still achieved and velocity consensus is reached.

However, when Algorithm 1 is applied, the agents achieve the desired formation and reach velocity consensus. Additionally, the adversarial inputs are attenuated in accordance with equation 2.14, as shown in Figure 2.6.

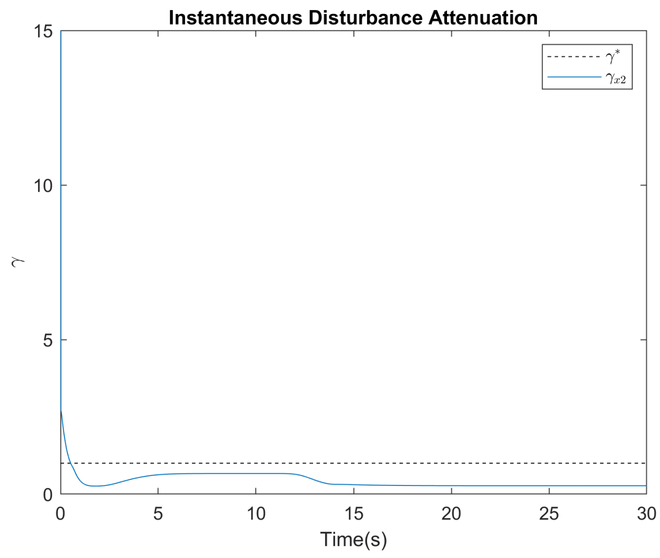


Figure 2.6. Instantaneous disturbance attenuation for the agent subject to attack. Notice that once learning is complete the attenuation remains below the specified H_∞ gain.

Figure 2.7 shows that the critic's neural network weights converge to constant values and remain stable once learning is complete and the adversary is attenuated.

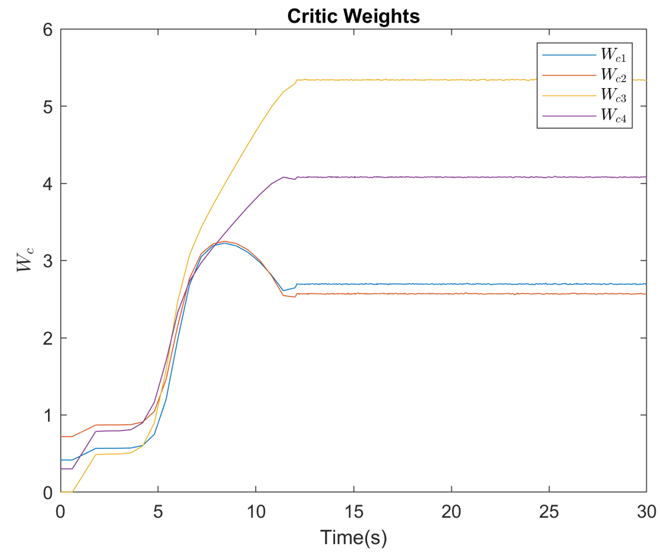


Figure 2.7. Time history of weights for the critic neural network approximator.

A demonstration of Algorithm 1 was also implemented in the Gazebo robotics simulator for enhanced realness in the form of software-in-the-loop simulations. This further validates the method proposed here and reduces the technological gap in implementing the proposed method in real physical systems, which is the ultimate goal. Implementing in real hardware is a direction of future work. Here, we have a large swarm of 31 drones charged with tracking in a formation resembling the Purdue logo. The smaller inlay frames only help to visualize the formation shape. The left frame shows the stabilizing distributed formation control algorithm alone, and the right frame shows the inclusion of this learning-based resilience method under the same conditions.

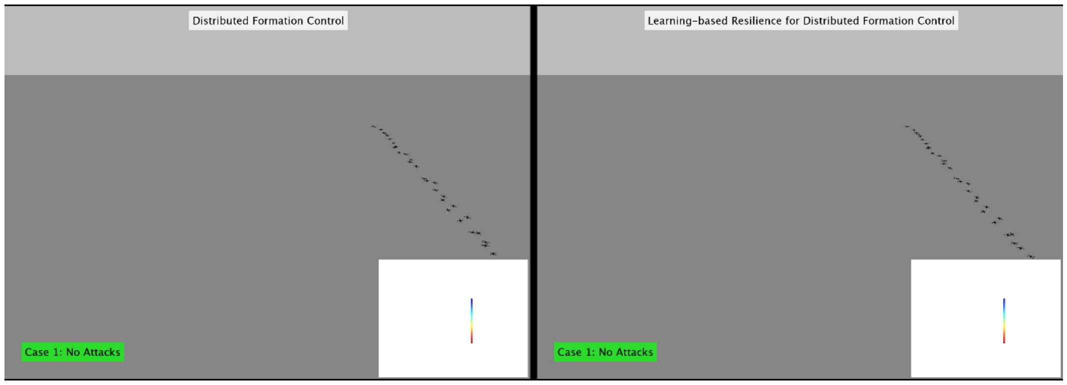

Distributed Formation Control
Case 1: No Attacks
Learning-based Resilience for Distributed Formation Control
Case 1: No Attacks

Figure 2.8. Gazebo simulation initial conditions.

First, no adversarial inputs are given and we observe that both achieve proper formation tracking. The learning-based resilience assembles slower since the neural network weights are initialized to random values rather than zeros only to show that the weight tuning laws also appropriately drives the compensating input to zero when there are no attacks.

Next, a persistent sinusoidal attack is launched on only the first follower dispersing both formations. It is subject to the adversarial inputs $u_a = \begin{bmatrix} 3\cos t + \sin 3t \\ \cos 3t + 3\sin t \end{bmatrix}$. Even though only one agent is under attack in this large formation, it completely disrupts all agents.

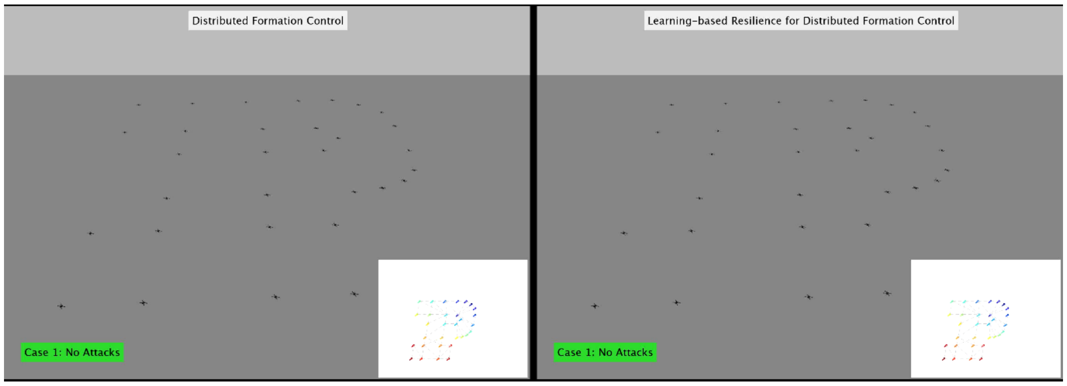

Distributed Formation Control
Learning-based Resilience for Distributed Formation Control
Case 1: No Attacks
Case 1: No Attacks

Figure 2.9. Gazebo simulation with no attacks. Both methods achieve formation tracking.

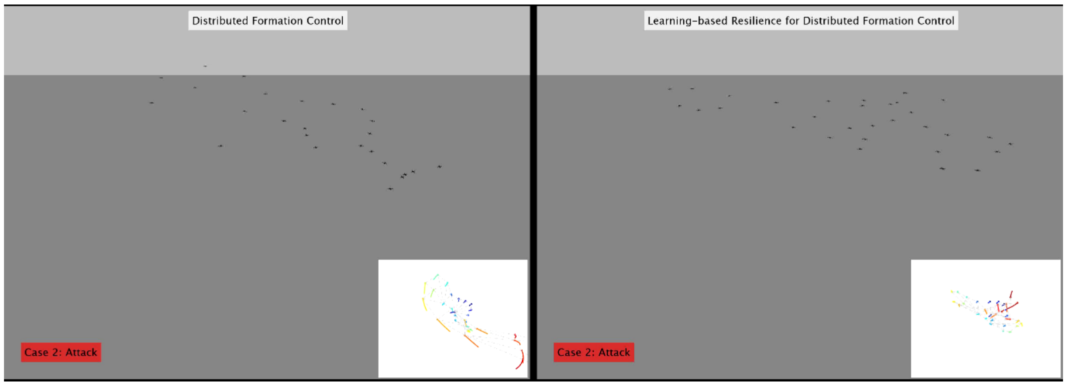

Distributed Formation Control
Learning-based Resilience for Distributed Formation Control
Case 2: Attack
Case 2: Attack

Figure 2.10. Gazebo simulation of first follower under attack. Both formations disperse when attack is launched.

In the left frame, the attack continues to disrupt the formation unceasingly, while the learning-based resilience method allows the formation in the right frame to attenuate the adversarial inputs and recover to proper formation assembly and velocity consensus.

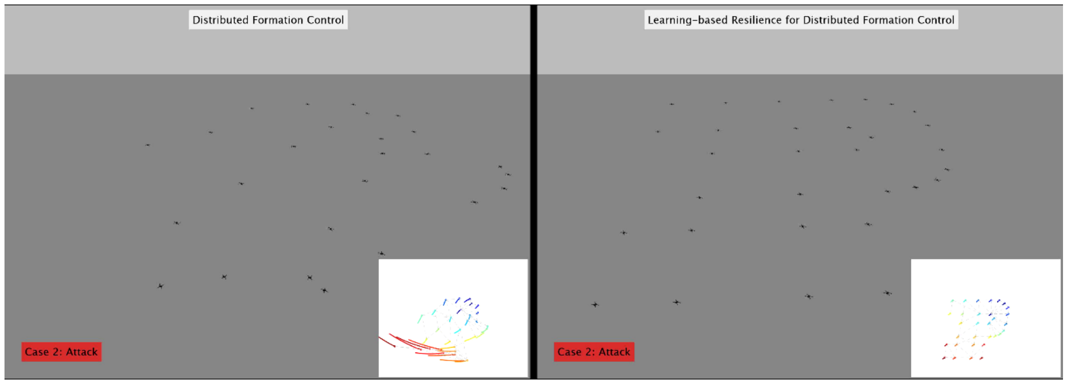

Distributed Formation Control
Learning-based Resilience for Distributed Formation Control
Case 2: Attack
Case 2: Attack

Figure 2.11. Gazebo simulation of first follower under attack. With our learning-based compensation, the formation recovers and achieves tracking.

3. BEHAVIOR-BASED CYBER-PHYSICAL ATTACK DETECTION

3.1 Introduction

While in the previous section an attack-resistant control scheme was developed for multi-agent systems, it is still important for the operator to have awareness when such attacks take place. This falls into the realm of attack detection, which has become an increasingly popular research area in recent years as adversaries develop more and more sophisticated attack surfaces. Motivated by this observation and the goals of our sponsors, we sought to devise a method of detecting attacks on autonomous air vehicles with a broad scope, as many previous studies offer reactive solutions based on attack signatures which do not provide any zero-day protection. At the same time, we notice machine learning methods gaining traction in related fields like anomaly/intrusion detection for smart infrastructure and computer networks. Therefore, we seek to emulate the success other fields have found in counter-autonomy defense here.

3.2 Monitoring System

The main goal of this work is to consider the feasibility and effectiveness of a behavior-based machine learning model which can learn normal UAV system operation and recognize behavior which is likely occurring due to an attack. To accomplish this, the model will act as a monitoring system similar to a sanity check. The figure below depicts a block diagram of a UAV controller represented as a cyber-physical system.

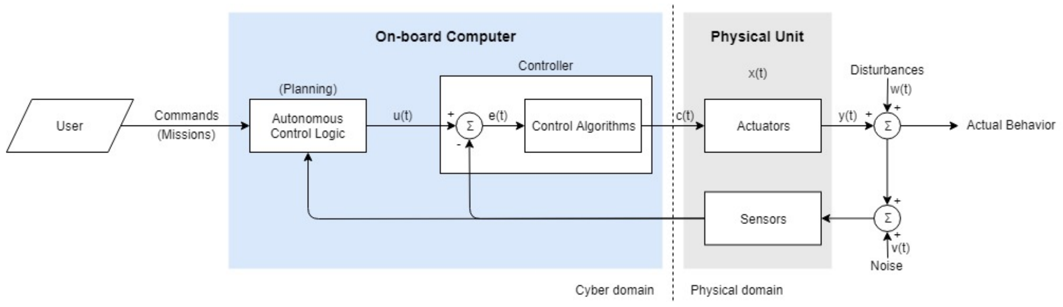


Figure 3.1. Typical UAV control system.

Here, we propose an additional block which utilizes only the data already available and processed by the control system, which reduces the barrier-to-entry for such a system since other methods might require extra sensory data. This new block will be the trained monitor, resulting in the modified block diagram below.

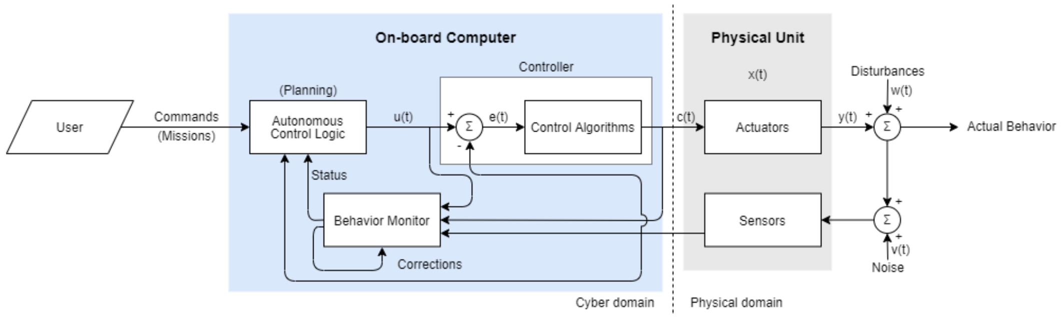


Figure 3.2. UAV control system with monitor added.

As depicted, the monitor program resides in the cyber domain and acts as a check for interactions between the cyber and physical systems. Its predictions are used by the high-level control logic to assess the status of the entire vehicle. This ability enables operators to have real-time warning of ongoing attacks and vulnerabilities in their system. While some methods of response to a detected attack have been suggested, such as back-tracking as in [26], this work is restricted to the detection only.

Training will consist of two phases. First, off-board learning will be accomplished by training a machine learning SVM classifier on a dataset. This model establishes reasonable delineations between the attack behavior data points and the nominal behavior data points. This provides a starting point for the monitoring system and is usually computationally expensive. Once training is performed, however, predictions on new data are very quick and efficient. Still, to constantly improve the model during operation, learning does not end there. On-board learning will be utilized in the form of corrections by operators. This is depicted in Figure 3.2 as the loop originating and terminating at the monitor. This allows the model to adapt to new behaviors which might be required of the system. This is achieved by leveraging the delineation created during the off-board training. The operator establishes correction parameters in the form of a maximum distance from the decision boundary and the number of consecutive points within that distance. As the system is operational, new data points are presented to the model. If they lie within the region defined by those parameters, manual checks are elicited for human correction. As such, a trade-off exists between the ability to tweak the boundary during on-board learning and the number of manual corrections elicited. The optimal balance will likely depend on the particular application, but we will show that such a method is successful in improving model performance.

3.3 Simulated Cyber-physical Attacks

In order to establish the relevance of this work, the integrity of the data must be shown. To accomplish this, the methods of data generation must be detailed. This section elaborates on the methods of simulation which make up the resulting dataset. [26] provides a thorough examination of the privacy and security challenges facing UAV systems. Among these is the identification of cyber-physical threats to UAVs. Namely, these threats exploit physical channels and components on the vehicle to alter their operation (e.g. crash or redirect). The authors note that cyberattacks are potentially easier to detect, and adversaries are starting to focus on physical components. Because of this, we focus on cyber-physical attacks as well, since they are highlighted as current vulnerabilities. Potentially the most well-known cyber-physical drone attack is sensor spoofing, where an outsider feeds faulty information to sensors in order to alter their operation. In 2011, the Iranian military used such an attack to capture an advanced reconnaissance RQ-170 drone operated by the CIA. Attackers perform this spoofing usually through a combination of jamming the nominal signal (e.g. true GPS broadcast) and transmitting the malicious signals with much larger power. Other cyber-physical attacks like control signal spoofing is accomplished via powerful acoustic transmitters capable of modifying signals in transit from the bus to the actuators. While some software-based methods have been proposed to defend against attacks of this class, they are mostly highly specialized and signature-based. [26] highlights concern about this, since signature-based solutions off no protection against zero-day attacks and are always one step behind the attackers. Recently, [29] proposed a promising software-based solution to the same set attacks using a specification-based approach. However, specification-based approaches require intricate understanding of the underlying physical processes which can be difficult to obtain and sensitive to dynamic operating environments, ageing, or other evolvements of the system. Here, we propose a behavior-based detection scheme which utilizes machine learning techniques, similar to that demonstrated in a smart resource grid context [31]. This approach requires no a priori construction of specifications or formal system models. Rather, it automatically learns key behavior characteristics through operational data in order to assess system status. The resulting model is easy to develop (especially in the case where historical data is readily available), can make predictions quickly, and adapt evolving system behaviors. This is accomplished using a combination on supervised and unsupervised learning. Whereas, unsupervised learning methods do not require large sets of labeled data for training but are extremely prone to false positives or missed detections, supervised learning can be much more accurate but requires labeled data. We propose a model training phase via supervised learning followed by unsupervised learning for increased model adaptability.

The main challenge to utilizing supervised training is the requirement for labeled training data. Since comprehensive data from real-world examples of the attacks considered here are not readily available and the equipment required to perform them is costly and difficult to develop, we must rely on data generated by simulations. Luckily, these types of attacks can be simulated accurately. These attacks can be categorized into three main groups: control signal spoofing, sensor spoofing, and parameter corruption. The authors in [58] point out that these classes of attacks lack research attention and defense methods.

All simulated data was gathered using a Gazebo simulation environment for an Iris+ quadcopter using a PX4 autopilot. Gazebo is a 3D robotics and physics simulation environment capable of a high level of fidelity. The Iris+ model and PX4 plugins are available open source from the PX4 developers. Interaction with the PX4 (and, hence, the vehicle) is done via MAVlink messages. The spatial environment is defined by a safe zone of airspace on Purdue's athletic fields. A real world terrain map is used as ground level and a 0.5km ceiling was defined. The simulation process consists of high-level mission planning and low-level operational parameter planning. That is, each simulated mission is composed of several high-level tasks like taking off, navigating to a waypoint, loitering, and landing which each have associated operational parameters like waypoint position, loiter time, etc. This delineation allows formulation of a state machine which captures the possible high-level behavior of the vehicle, as shown in [29]. For each simulated mission, a random walk through the state transition diagram shown below generates the sequence of operations the drone will accomplish. It is important to note that while this state machine captures rudimentary behaviors which might be typical of some UAV applications like delivery, the idea behind it could be applied to any application in which the high-level tasks can be captured in a similar state transition diagram. Low-level parameters specifications are accomplished by first establishing limits on each parameter (usually stemming

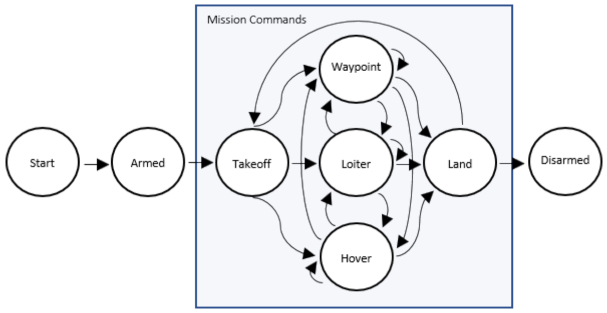


Figure 3.3. State transition diagram for mission generation.

from operating limitations like max airspeed, max roll angle, etc.) and then choosing values via random sampling from a uniform distribution across those limits.

These plans are compiled into executable instructions via MAVlink commands, which are uploaded to the drone before each simulated mission using QGroundControl, a GUI for interacting with the PX4. The PX4 autopilot then executes the missions autonomously and logs all computed data, just as it would in real hardware. The only difference in simulation is the PX4 directs the Iris+ Gazebo model and plugins and Gazebo in turn updates the world state, rather than directing real flight hardware. Sensors measurements are simulated based on the Gazebo world state and physics simulation. The logs kept by the PX4 autopilot will be used to describe the drone's behavior. Altogether, 95 missions were simulated: 35 nominal and 20 of each type of attack. Five missions of each type were withheld from the model during training to evaluate classification performance.

3.3.1 Control Signal Spoofing Attacks

To simulate control signal spoofing, the motor control pulse width modulation (PWM) signal is targeted, as in [29]. This signal is used to adjust and control the motors' rotation. This particular signal was chosen here because its criticality makes it a likely target for manipulation: if attackers can control the motors, they can control the operation of the vehicle. In fact, this type of attack is exploited in [58], [59], and [60].

In simulation, the PWM signal is modified completely externally as it would be in the three instances above. At the interface between the PX4 autopilot (simulating the vehicle control system) and Gazebo (simulating the vehicles environment), the width of the PWM signal is altered, resulting in improper RPM settings for the motors. PWM is a method of achieving analog characteristics from a digital control signal. In this case, the motor receives control signals in pulses at a fixed voltage. The length of the pulse, as a proportion of the specified maximum pulse length, determines the RPM setting of the motors, as a proportion of their maximum duty cycle. For instance, here, PWM signals are generated with an amplitude of 3.3V and a maximum pulse duration of 2.5 ms. A pulse sent to the motors with amplitude 3.3V and duration of 1.25 ms would command the motor to 50% of its maximum RPM setting.More specifically, PWM uses a rectangular pulse wave with modulating width to communicate the desired duty cycle to the motor's speed controller. The motor directly measures this pulse width to determine the output power level corresponding to the desired duty cycle. For the PX4, the pulse width in milliseconds as a function of desired duty cycle is

$$W(k) = 2.5D(k) \tag{3.1}$$

where $D(k)$ is the desired duty cycle calculated by the autopilot at discrete time step k. To simulate control signal spoofing, a malicious bias signal width is added at each time step.

$$W'(k) = W(k) + \alpha \tag{3.2}$$

The resulting signal, $W'(k)$, must be within the PWM limits to have an effect on the motor, so $\alpha \in [-W(k), 2.5 - W(k)]$. To add generality to the resulting dataset since there is no knowledge of the exact intent behind the spoofing attack, α should be chosen with some level of randomness so as to not be trivially detectable but not completely arbitrary on its domain since some attacks might not require large deviations. For each mission a mean bias, b, is generated for the manipulating signal. As such, Equation 3.2 is rewritten as

$$W'(k) = W(k) + \alpha,\ \{\alpha \mid \alpha \sim N(W(k) + b, \sigma_\alpha^2) \wedge \alpha \in [-W(k), 2.5 - W(k)]\} \quad (3.3)$$

where the variance of the truncated normal distribution, σ_α^2, is chosen for each mission to simulate varying levels of randomness. Smaller variances result in spoofed signals nearer to the controller signal width, $W(k)$, while large variances approach a uniform distribution which would be complete randomness. For training, the goal here is to simulate what might constitute abnormal behavior, not just a specific attack. The test missions were subject to specific attacks like control signal saturation. These are more evident of what attacks may exactly look like, and they provide us with an opportunity to see if the trained model might recognize them from only learning on more subtle attacks. This alteration occurs at the interface to the motor's Gazebo plugin, as shown in the figure below. Injection of the malicious signal at this point simulates

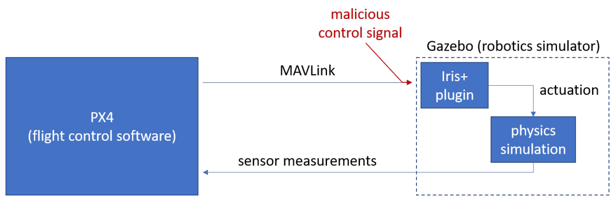


Figure 3.4. Software-in-the-loop simulation flow for control signal spoofing attacks.

tampering in transit from the vehicle's bus to the actual motor. The resulting motor function is captured and modeled by Gazebo, and sensor measurements are logged as normal. Only a single, randomly-selected motor is targeted in each mission simulating this attack. Each attack prevented normal operation as planned (optimal trajectory to waypoints, etc.) and some even resulted in ground collisions (denial of service).

3.3.2 Sensor Spoofing Attacks

Sensor spoofing attacks are possibly the most popular ones that exploit physical channels on the vehicle. Attackers use external equipment to force false readings at certain sensors, which could drive behaviors toward their desired state. GPS ([61–63]) and IMU ([63–65]) sensors have been shown to be most vulnerable, so these were the targeted sensors for simulation here. Either GPS or IMU readings were altered exclusively on missions, never both together. The same methods were followed as in [29], where the measured values from the sensors in Gazebo were modified at the interface between Gazebo and the autopilot. In missions where GPS was targeted, the GPS position and velocity measurements were altered. Likewise, in the missions where the IMU was targeted, the accelerometer measurements were altered. The accelerometer was chosen because several sources identify it as a vulnerability [64,65].

The intent of a sensor spoofing attack must be considered in order to properly simulate it. If an attacker were simply seeking to cause these sensors to fail, it might alter the measurements randomly. It seems more likely, however, that an attacker would seek to alter the signal to drive guidance in a certain direction. Therefore, in each mission simulated, a random direction (represented by the unit vector $\hat{u}_{attack}$ in the Earth-centered coordinate frame) was chosen as the target direction, which stayed constant for the entire mission. Altering sensor measurements to drive the vehicle in this direction would include feeding the autopilot false readings in the opposite direction so that it compensates toward the attacker's desired destination. Therefore, each measurement is altered with some bias opposite of the attack direction according to the following equations which incorporate some randomness to avoid static bias which might be easier to detect. For GPS, the spoofed position, $x'(k)$, and velocity, $v'(k)$, measurements are given by

$$x'(k) = x(k) - 2x_b R\hat{u}_{attack} \tag{3.4}$$

$$v'(k) = v(k) - 2v_b R\hat{u}_{attack} \tag{3.5}$$

where $x(k)$ and $v(k)$ are the nominal position and velocity readings in the Earth-centered coordinate frame, x_b and v_b are the bias center magnitudes (here 5 m and 2 m/s, respectively), and $R \sim U(0,1)$. Spoofed accelerometer readings, $a'(k)$, are given by

$$a'(k) = a(k) - 2a_b R\hat{u}_{attack} \tag{3.6}$$

where $a(k)$ is the nominal accelerometer measurement in the Earth-centered coordinate frame, and a_b is the bias center magnitude (here 0.5 m/s^2).

These signal alterations occur after sensor measurements are taken nominally in Gazebo but before they are received by the autopilot, as shown in the figure below. Modification at this point simulates external tampering as it might occur in real

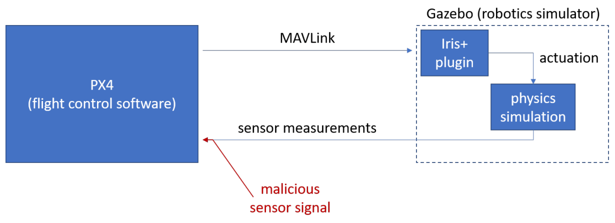


Figure 3.5. Software-in-the-loop simulation flow for sensor spoofing attacks.

attacks. Each attack prevented normal operation as planned (optimal trajectory to waypoints, etc.) but none resulted in ground collisions.

3.3.3 Parameter Corruption Attacks

Parameter corruption attacks are perhaps the simplest to simulate, since they involve only modifying normally-constant parameter values that impact operation. As in [29], the gains of attitude rate PID controllers were chosen as the focus of these attacks since their ability to control motor function makes them likely targets. In [66], parameters like these were shown to be easily altered, and [67] even used only localized heating to modify parameter values like this, which could be accomplished from a distance. These values are often tuned for desired performance and largely left unchanged during normal operation. Three PID controllers individually handle body rates in each axis on this vehicle. Each PID controller consists of three coefficients which govern its behavior. The proportional gain is used to minimize tracking error, the derivative gain is used for dampening, and the integral gain keeps memory of the error. Modification of these gains will affect the controllers' handling of the body rates. Each simulated parameter corruption attack exploited all gains for one randomly-chosen controller governing an axis. The actual values of these gains are withheld from the monitor model to prevent superficial checks and assess its ability to recognize this attack from the vehicle's behavior.

Before each mission, a random sample from the uniform distribution over the limits of each gain (given in the PX4 developer's guide) was evaluated and set as the modified gain. The parameters were changed directly in the flight control software's source code, as illustrated in the figure below. Each attack prevented normal operation as

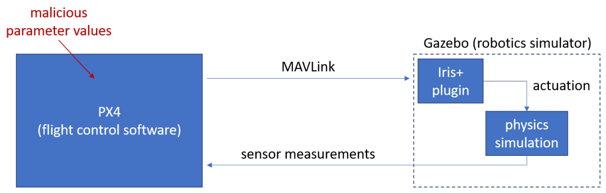


Figure 3.6. Software-in-the-loop simulation flow for parameter corruption attacks.

planned (optimal trajectory to waypoints, etc.) and some even resulted in ground collisions (denial of service).

3.4 Classification

We formulate the classification problem as a binary discrimination one where the classifier must find the best hyperplane delineation in the feature space. More specifically, the classifier seeks to output a $status \in \{nominal, attack\}$ of the vehicle. Altogether, there are 174 features recorded in the PX4 log files. Therefore, we have a high-dimensional dataset. The SVM classifier is capable of exploiting the so-called "kernal trick" to find a suitable delineation without debilitating increases in computational expense. The SVM output score also provides a convenient metric of the distance to the decision boundary for use in eliciting manual corrections. Figure 3.7 illustrates a simple SVM used for binary discrimination.

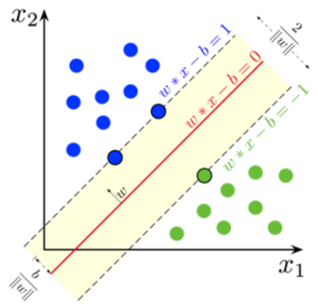


Figure 3.7. Example 2-D SVM.

In order to evaluate the model, we must describe performance measures. There are a handful of scenarios for each data point based on the model prediction and the ground truth label. Each data point has a ground truth label which the model's classification is compared against. If the model's classification matches the ground truth, this is known as a *true* classification. If the model's classification does not match the ground truth, it is known as a *false* classification. The *attack* class is chosen to be the *positive* one. This means that classifications of an attack are called *positives*, and classifications of nominal are called *negatives*. Combining these terms, *True positives* are data points correctly classified as *attack* by the model, *true negatives* are data points correctly classified as *nominal* by the model, *false positives* are data points incorrectly classified as *attack* by the model, and *false negatives* are data points incorrectly classified as *nominal* by the model. Figure 3.8 illustrates the different possibilities and how they relate to model evaluation.

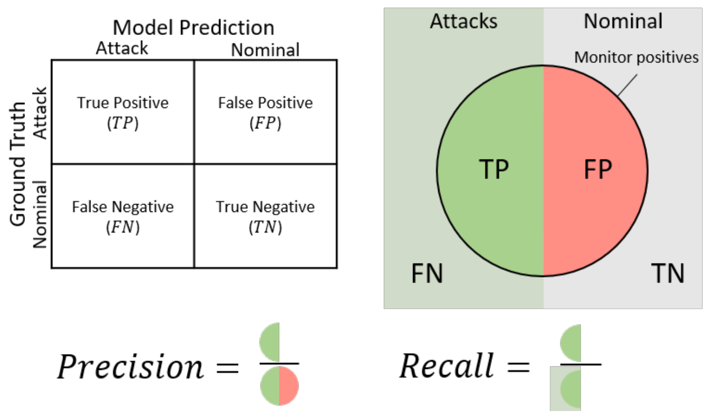


Figure 3.8. Performance metrics for binary discrimination.

The two main applicable performance metrics are *precision* and *recall*. Precision is most intuitively described as the percentage of attack data correctly classified by the model that is truly attack data. This relationship can be written as

$$Precision = \frac{TP}{TP + FP}. \tag{3.7}$$

Similarly, recall is most intuitively described as the percentage of true attacks that were correctly predicted. This is written as

$$Recall = \frac{TP}{TP + FN}. \tag{3.8}$$

Initially, we found model performance to be noisy and subpar, so further improvements were required to increase accuracy and reduce false alarms.

3.4.1 Temporal Context Features

Using the 174 features available from the flight logs, the SVM classifies each data point independently. Initially, this resulted in subpar performance. As Table **??** shows, direct classification by the SVM only yielded 88% precision and 73% recall. Therefore, we sought a way to improve model accuracy and reduce nagging false alarms. Recognizing that there is likely useful information in the relation between features at different time steps which is unavailable to model, we require a way of including temporal context for each of the features. Perhaps the simplest way of accomplishing this would be to include the features from previous data points with the current features at each time step. This is known as the sliding window method. However, depending on the amount of past data considered, this would drastically increase the number of features and does not scale well in terms of training time, classification time, and computational burden. In fact, if w is the window length, the number of features would be increased to $174 \times w$, which does not scale well with an increasing amount of temporal context available to the model. We propose using an augmented sliding window technique, in which only the mean and standard deviation of the past w points are kept. This restricts the number of features to 174×3, no

$$y(t_N) = f\Big(\underbrace{x_1(t_N), x_2(t_N)}_{\text{Current features}}, \underbrace{\bar{x}_1([t_{N-w}\ t_N]), \bar{x}_2([t_{N-w}\ t_N])}_{\substack{\text{Mean of each feature} \\ \text{from past } j \text{ steps}}}, \underbrace{s_{x_1}([t_{N-w}\ t_N]), s_{x_2}([t_{N-w}\ t_N])}_{\substack{\text{Standard deviation of each feature} \\ \text{from past } j \text{ steps}}}\Big)$$

Figure 3.9. Augmented sliding window technique, where N is the data point index.

matter the size of the sliding window. Inclusion of temporal context should improve the model's recall performance.

3.4.2 Kalman Filter Post-processing

As noted previously, the SVM classifier makes a decision on every data point, which is done at the high logging frequency. This produces a noisy signal which causes spurious false alarms. To reduce the number of false alarms, it is desirable to devise a method of estimating the true state of the vehicle based on the cumulative successive outputs of the SVM. One such method is utilization of the Kalman filter on the SVM classifications. We implement a second order discrete time Kalman filter which estimates the true state of the vehicle, x_k, based on noisy outputs of the SVM, y_k, while assuming the true state rate of change is stable with smooth transitions and the measurement noise is much larger than the process noise. This filter can be described by the state space model below, where the relation $\sigma_v^2 \gg \sigma_w^2$ can be tuned for the desired level of filtering.

$$X_k = \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix} \qquad w_k \sim N\left(0, \begin{bmatrix} {}^{\Delta T^4}\!/_4 & {}^{\Delta T^3}\!/_2 \\ {}^{\Delta T^3}\!/_2 & \Delta T^2 \end{bmatrix} \sigma_w^2\right)$$

$$X_{k+1} = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} X_k + w_k \qquad v_k \sim N\left(0, \sigma_v^2\right)$$

$$y_{k+1} = \begin{bmatrix} 1 & 0 \end{bmatrix} X_{k+1} + v_{k+1}$$

3.5 Results

We observed promising results when applying the model to our test dataset. Table 4.1 summarizes the results for each mission. Notice that the model correctly identified each of the attacks in the test dataset and did not output any false warnings. The mean warning time was about two seconds, which would likely be quick enough to initiate a defensive response, depending on the attack type/severity. The warning time is directly related to the measurement noise parameter of the Kalman filter. Smaller assumed measurement noise would force the model to respond more quickly to *attack* SVM classifications. However, this comes with added risk of more false alarms. Therefore, operators should take care to tune this relationship based on the allowable risk.

Table 3.2 shows the dramatic performance increase associated with including temporal context and the Kalman filter post-processing. These two methods are key design features that enable effective status assessment by the monitor.

Figure 3.10 shows the dramatic increase in performance when even a small amount of temporal context is included, but the improvements plateau beyond window lengths of more than about one second. While the inclusion of artificial temporal context features does increase the number of predictors for the SVM, there was no significant increase in supervised training time.

Table 3.1. Summary of test dataset results.

Test Mission No.	Attack Type	Predicted Status	Warning Time (s)	False Alarms
1	None	Nominal	-	0
2	None	Nominal	-	0
3	None	Nominal	-	0
4	None	Nominal	-	0
5	None	Nominal	-	0
6	Control Signal	Attack	2.01	0
7	Control Signal	Attack	1.53	0
8	Control Signal	Attack	0.57	0
9	Control Signal	Attack	3.02	0
10	Control Signal	Attack	3.87	0
11	GPS Signal	Attack	1.56	0
12	GPS Signal	Attack	2.44	0
13	GPS Signal	Attack	3.62	0
14	IMU Signal	Attack	3.61	0
15	IMU Signal	Attack	1.83	0
16	Roll PID	Attack	0.79	0
17	Roll Rate PID	Attack	1.42	0
18	Pitch PID	Attack	3.24	0
19	Pitch Rate PID	Attack	0.47	0
20	Yaw Rate PID	Attack	1.55	0
Mean			2.10	0

The impact of Kalman filtering the SVM classifications can be seen in the model output from the test mission in Figure 3.11. It is particularly beneficial in preventing false alarms when the SVM classifications are near the decision boundary, but a conclusive spike has not yet been observed. This has the effect of reducing manual correction elicitations and provides operators with a clearer picture of the vehicle's

Table 3.2. Effect of temporal context features and Kalman filter post-processing.

	SVM Direct	Temporal Context, No Kalman Filter	Kalman Filter, No Temporal Context	Temporal Context, and Kalman Filter
Precision	88.2%	94.3%	90.2%	98.1%
Recall	73.1%	84.7%	75.6%	88.4%

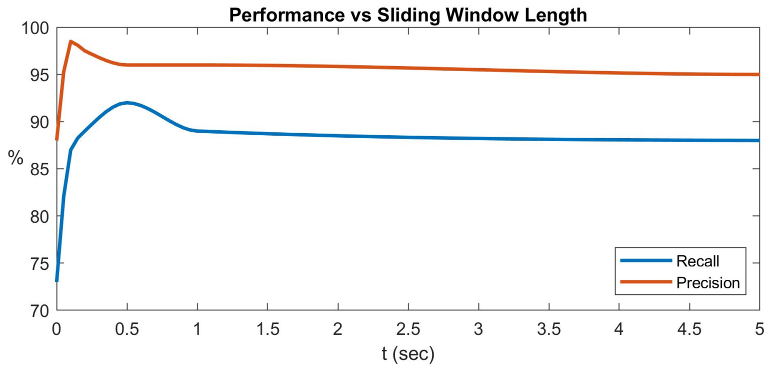


Figure 3.10. Performance versus sliding window size.

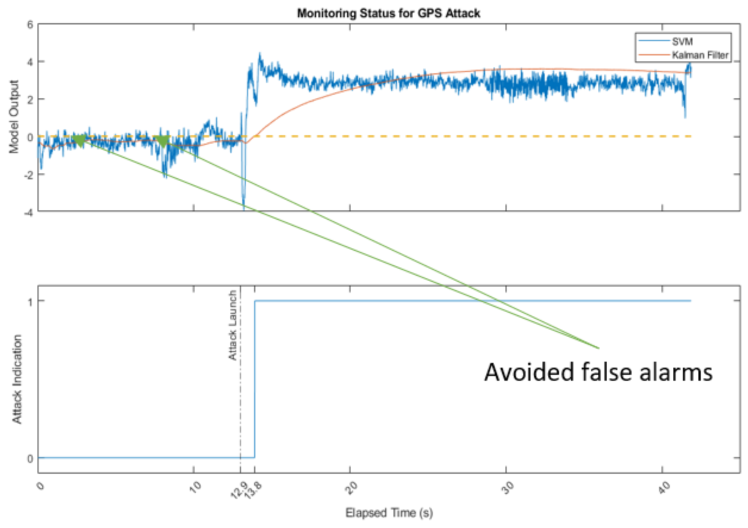


Figure 3.11. Effect of Kalman filtering on model output.

status. In all cases and window lengths, the filtering improved precision by at least a few percent.

On-board learning by manual corrections proved effective in improving recall, but only marginally. While a more comprehensive study in terms of flight data and correction incorporation would benefit the field, the results of this work suggest that these corrections do in fact improve performance of such a behavior monitor. Figure 3.12 shows the recall improvement observed for varying correction parameters. The

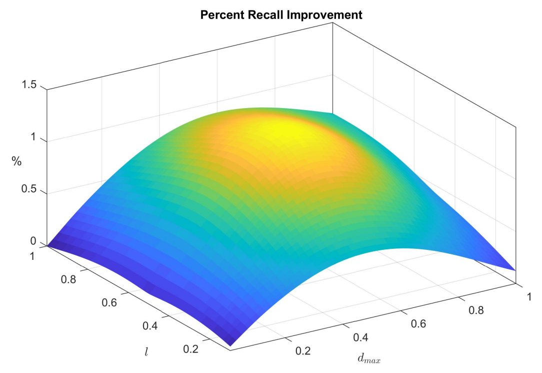


Figure 3.12. Effect of correction parameters on model performance.

relationship between improvement and parameters will likely be largely dependent on the specific application. Still, this work suggests that such a method is useful in improving performance during operation, allowing the system to keep learning after training. This is important, since interaction between autonomous systems and their operators is a critical dynamic, and it is one that, when mastered, will truly unlock the full potential of autonomous systems. Future work should include creation of an operator-system interface which streamlines the correction process and further reduces technical burden on the operator.

4. EPILEPTIC SEIZURE PREDICTION USING A HYBRID METHOD OF MACHINE LEARNING TIME SIGNAL ANALYSIS

4.1 Introduction

In the previous section, a method of analyzing time signal data using machine learning was developed which consisted of a traditional machine learning mechanism (the SVM) and augmenting techniques including artificial temporal context features and Kalman filter post-processing. These augmentations proved critical in boosting the performance of the overarching system. We also note that time signal analysis has extremely broad applications, particularly in fields related to autonomy, such as computer vision, sensor fusion, telemetry analysis, etc. Motivated by this, and a collaboration request from a Yale neuroscience research group, we were presented with an opportunity to generalize our hybrid method to other datasets, further examining its potential for other applications. The collaborators at Yale seek a system for predicting seizures in epileptic patients based on signals collected by electroencephalography (EEG). In this section, we combine traditional feature extraction methods in the form of deep convolutional neural networks (CNNs) with our classifier techniques developed in the previous section to form a hybrid method of time signal analysis using machine learning.

4.1.1 Epilepsy

Epilepsy is a brain condition characterized by chronic, recurrent, unprovoked seizures. It affects one in every 26 Americans, and places a significant risk of injury and other complications on those afflicted [38]. Patients are exposed to significant risk of harm during these seizures, which negatively impacts their quality of life since their seizures are spontaneous and, hence, their safety is constantly uncertain. Seizures can cause coincidental accidents like drowning or injury, but the risk of Sudden Unexpected Death in Epilepsy (SUDEP) is also an important concern. SUDEP is the leading killer of people with uncontrolled seizures [36]. Its cause is not known, but it may [68, 69] be due to seizure-induced apnea. In these cases, death can be avoided by simply physically stimulating the patient to restart the normal breathing process [68]. While therapies and medications do exist, many times the patient's risk factor can be dramatically lowered by simply assuming a safe position/environment and being monitored by an able caretaker. As such, a system capable of providing advance notice to patients and caretakers would prove extremely valuable, both in terms of decreasing the patient's risk and improving their quality of life. Upon successful prediction, the patient could take precautions to reduce environmental risk, alert a caretaker for proper supervision, or even medicate to suppress the impending seizure altogether.

4.2 Dataset

The dataset used here comes from experimental data collected at Boston Children's Hospital, which was made public from the study in [70]. It consists of EEG recordings from 22 patients of the hospital, both male and female between the ages of 1.5 and 22. 23 EEG electrodes were placed at specified locations on the scalp, and the signals were recorded at a sampling frequency of 256 Hz. While some recordings lacked all 22 electrode locations due to physical limitations during the experiment, we restricted the dataset to only those with the same amount of electrodes and corresponding scalp locations. Both interictal (between seizures, normal brain activity) and ictal (during seizure) data were included. All seizures were manually verified by video recording and labeled accordingly. In total, the dataset used here consisted of 198 seizures and 209 interictal hours. From these records, three partitions were created. 70% of the data was used for training the model (training set), 15% was used for validation during training (validation set), and 15% was used to evaluate the model performance (test set). Each of these partitions consisted of proportionally equal parts interictal and ictal data.

4.3 Method

In order to predict imminent seizures, the developed must recognize preictal (just before seizure onset) brain activity as recorded on the EEG which is reliably characteristic. Therefore, the underlying objective is to create a machine learning model which can accurately distinguish between interictal and preictal brain activity. Figure 4.1 below illustrates the method we propose to accomplish this objective. First, the

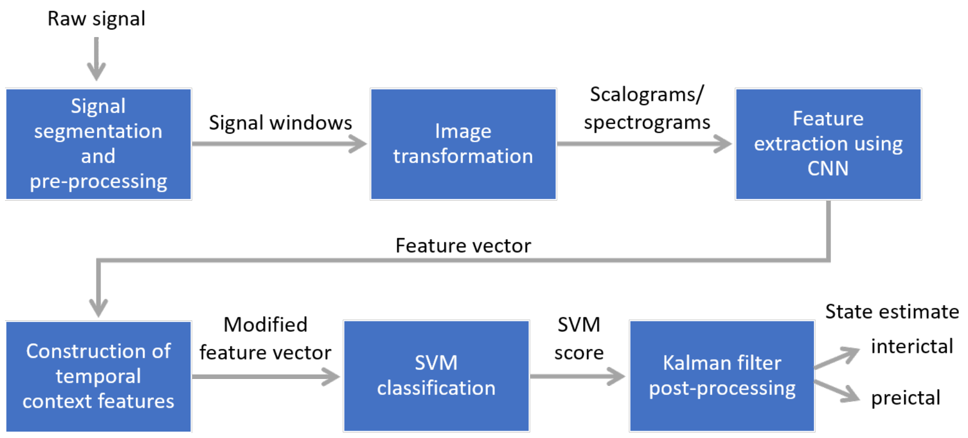


Figure 4.1. Visual representation of seizure prediction process by the proposed method.

raw EEG signals are segmented into labeled windows. Each window is then transformed into a two-dimensional signal representation, which is used to train a deep convolutional neural network. A feature vector describing the learning features of the windows from the two considered classes (*interictal* and *preictal*) is extracted from the activations at the CNN's final layers. Then, the feature vectors from neighboring windows are used to create artificial temporal context features, which are included with the feature vectors to form augmented feature vectors. Next, the augmented feature vectors are used to train an SVM classifier. Finally, the SVM output scores are post-processed by a Kalman filter to produce an estimate of the true state of the patient.

4.3.1 Pre-processing

As with all deep neural networks, the quality and quantity of training data will significantly affect the model's performance. First, the raw signal must be properly labeled for our classification purposes. Figure 4.2 shows how one channel of a two hour section of signal is labeled for training. The labels provided with the dataset only

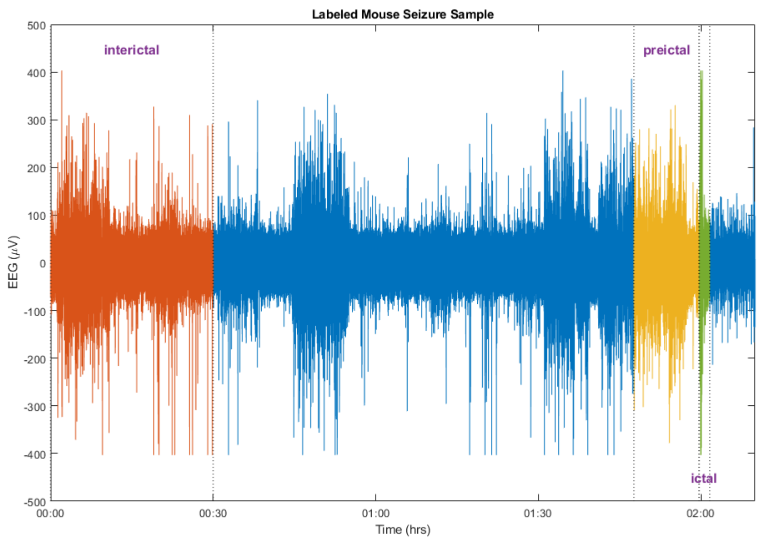


Figure 4.2. Labels for two hours of signal from a single channel.

identify seizures in the signal, and they provide no information on preictal or clean interictal activity. The model's ability to predict seizures hinges on the assumption that defining features of the preictal period will become present in the signal at some point before the seizure. For preictal activity to be recognized, its characteristic features must be present and the corresponding data labeled appropriately. This part of labeling is critical, since incorrect labeling will make training more difficult and degrade model performance. For example, consider a true preictal duration of 10 minutes prior to seizure onset. If we underestimate that the preictal duration is 4 minutes and label the data accordingly, the model will be incorrectly trained on 6 minutes of data that is labeled interictal but contains the preictal features. Similarly, overestimating the preictal duration as 14 minutes before seizure onset would result in incorrectly training the model on 4 minutes of data that is labeled preictal but contains no preictal features. Both scenarios would make it difficult for the model to distinguish between the interictal and preictal periods, which is a necessary ability. Further complicating matters, there is no identified point at which the preictal features become present, although other studies suggest it is usually around ten minutes before seizure onset [42]. We deal with this problem by defining a buffer period between the chosen preictal duration and the utilized interictal data for training. This is the blue region of signal in Figure 4.2. By refraining from using the buffer period data during training, we aim to ensure the model is being only being trained on data which has a high level of confidence in its label. As long as the buffer period is sufficiently large, it is unlikely that preictal features will be present in the interictal data used. Likewise, restricting the preictal period to only a matter of minutes before seizure onset ensures a high level of confidence that if preictal features exist, they will be present there. In Figure 4.2, the seizure occurs at two hours. Ten minutes before the seizure is used as preictal data, and interictal data is taken from signal at least 90 minutes before seizure onset. Large buffer periods are acceptable since the dataset is highly skewed in favor of the interictal class. That is, there is much more interictal data in the set than preictal. Therefore, large buffer periods can be used to balance the two classes.

The labeled data is then segmented into windows. This is necessary in order to consider data sections at frequencies other than 256 Hz and because CNNs learn features from two-dimensional inputs. The number of resulting windows of signal depends on the size of signal sections as well as the amount of overlap between windows.

Window overlap provides a convenient way of oversampling the dataset and forcing the model to refrain from learning time-local features. A trade-off exists between the computational cost associated with windowing, transforming, and training with a greater number of windows and the resulting potential resolution for seizure warning opportunities.

Image transformation of the windowed signal segments is done by two methods for visual representation of the signal. Fourier analysis produces two-dimensional spectrograms, which capture the amplitude of particular frequencies at the given time. Likewise, wavelet analysis produces two-dimensional scalograms, which capture the absolute value of the wavelet coefficients describing the signal at the given time. Both have advantages and disadvantages, so they were jointly considered here. Spectrograms are plagued by time/frequency resolution, where more accurate results in timing come at the expense of frequency, or vice versa. Scalograms, however, are not subject to this, but they do produce much larger images and are associated with a higher computational cost both during image transformation and CNN training, as the greater number of pixels requires a greater number of parameters to be learned by the model. Figure 4.3 shows an example of a scalogram and spectrogram from the same window of preictal signal. These images are generated for each labeled

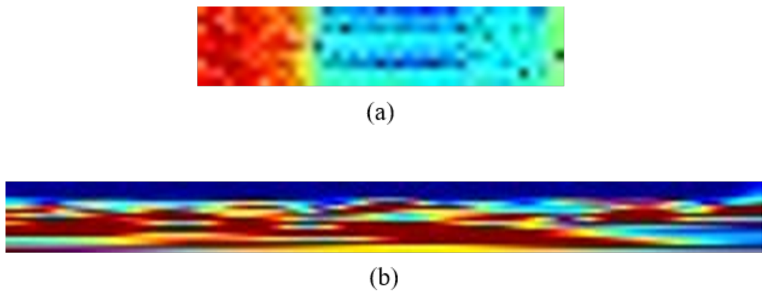


Figure 4.3. Example spectrogram (a) and scalogram (b).

window from each channel. All corresponding windows from each channel are then concatenated along a fourth dimension to produce a 3-D image representing the entire EEG signal. Therefore, each image has dimensions time, frequency, and channel, where the pixel values are in accordance with the visual representation (amplitude for spectrograms, absolute value of wavelet coefficient for scalograms).

4.3.2 Feature Extraction

Convolutional neural networks have proven effective in extracting features and classifying natural images in multiple instances [71, 72]. Therefore, they are likely to also be a good tool for classifying this signal data. An ideal model would undergo a training phase where it is trained on lots of labeled windows from the interictal and preictal periods. Once it has learned to distinguish between windows from each period, it could be deployed to analyze real-time EEG signal data and issue warnings when it starts classifying the current signal as preictal.

The optimal architecture of CNNs is a widely debated topic in the field, and it is largely thought to be highly dependent on the application. In order to be conservative, we use a proven approach for designing the network which consists of several blocks of convolution, batch normalization, and max pooling layers. This also allows us to directly compare our approach to the results of [41] in order to evaluate our model. Figure 4.4 shows the network architecture used here. The concatenated 3-D image

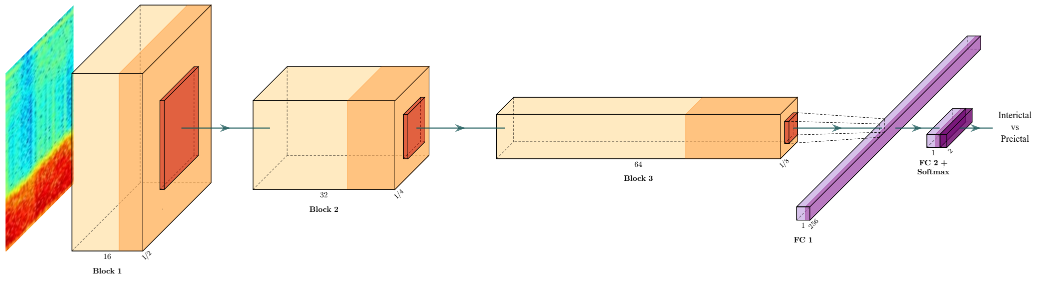


Figure 4.4. Convolutional neural network architecture used for feature extraction.

is fed into the first convolution block, which uses 16 3-D filters of size 5 by 5 by 23 (number of channels) with rectified linear unit activation (ReLU) and stride 2 by 2 by 1. These activations are then down-sampled by a max pooling layer over a region of 2 by 2 by 1 and passed to the next convolution block. Block two uses 32 2-D filters of size 3 by 3 with ReLU activation and stride 1 by 1. Again, these activations are then down-sampled by a max pooling layer over a region of 2 by 2 and passed to the next convolution block, which is identical to the second except for using 64 filters. After the third convolution block, the resulting activations meet a fully-connected layer (FC 1) consisting of 256 neurons. A final fully-connected layer (FC 2) with two neurons feeds a softmax activation layer for direct classification by the CNN. For our purposes, we are interested in a CNN which produces a feature vector representative of the input image rather than normal direct classification. Therefore, we use the 256 activations at FC 1 as the CNN feature vector. The activations here should capture the cascading features learned by the previous layers and convolution blocks, providing a convenient summary of the most relevant features in the signal. Hyperparameters associated with training like learning rate schedule, regularization, and momentum were tuned using Bayesian optimization. The validation data was used to monitor the progress of training. Training was concluded once the batch cross-entropy loss diverged from the loss on the validation set.

4.3.3 Temporal Context

Similar to the temporal context created in the previous section, we use neighboring feature vectors from the CNN to create temporal context features. The augmented sliding window technique, illustrated in Figures 3.9 and 4.5 below, will eventually provide the classifier with features that capture the recent behavior of the signal as well as what is currently seen. This should help the model draw better conclusions than it would if it was considering each window of data individually. The augmented sliding window technique produces an augmented feature vector of 768 elements which can then be classified.

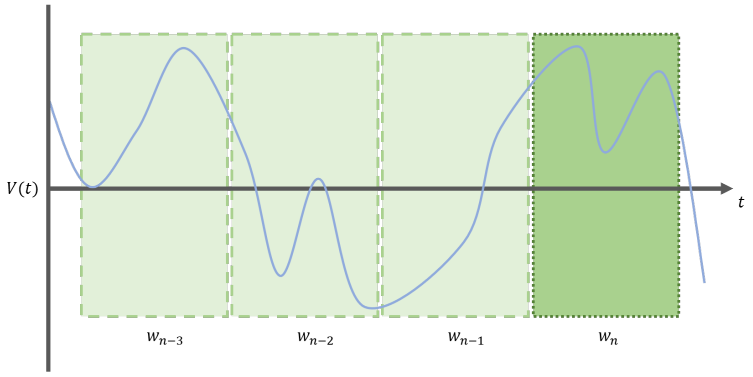


Figure 4.5. Augmented sliding window for temporal context. The current features are those from the dark green area, while the augmented feature vector includes the means and standard deviations of the features from the preceding windows.

4.3.4 Classification

We formulate the classification problem as a binary discrimination one where the classifier must find the best hyperplane delineation in the feature space between the augmented feature vectors of the interictal and preictal classes. More specifically, the classifier seeks to output a $status \in \{interictal, preictal\}$ of the patient. Altogether, there are 768 features in each augmented feature vector. Therefore, we have a high-dimensional dataset. The SVM classifier is capable of exploiting the so-called "kernal trick" to find a suitable delineation without debilitating increases in computational expense. Figure 3.7 illustrates a simple SVM used for binary discrimination.

Again, in order to evaluate the model, we must describe performance measures. Unlike the previous section, recall and precision will not be adequate, since we are more focused on metrics that evaluate the proposed system's impact on the patient's QoL. The authors in [40] provide a thorough examination of the different ways of assessing and comparing seizure prediction methods. Two such measures have become the standard means of evaluation: sensitivity and false alarm rate. Sensitivity is the percentage of seizures that the model correctly issues advanced warnings for, and false alarm rate (FPR) is the number of false warning issued per hour by the model when applied to the entire record. Still, we must further define what constitutes a correct warning. Following [41], we define a seizure prediction horizon (SPH), or the minimum time before seizure onset that a warning must be issued, of five minutes. We also select a seizure occurrence period (SOP), or the period of time after the warning and SPH that seizure onset must occur during, of 30 minutes. These two parameters are illustrated in Figure 4.6. For a warning to count as correct, a seizure must occur

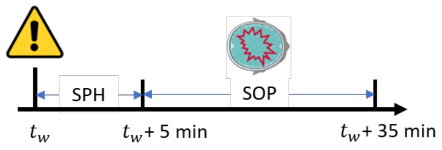


Figure 4.6. Window for correct warnings in terms of SPH and SOP.

no earlier than the SPH and no later than the SPH+SOP. Further, we compare our model to a random predictor for further validation. From [73], the probability of randomly issuing a warning in a given SOP with a given FPR can be approximated as

$$P \approx 1 - e^{-FPR \cdot SOP}. \tag{4.1}$$

Therefore, the probability, p, of predicting at least m seizures of M total seizures by chance is

$$p = \sum_{i \geq m} \begin{pmatrix} M \\ i \end{pmatrix} P^i (1-P)^{M-i} \tag{4.2}$$

In order to compare the performance of the model to the unspecified random predictor, p was calculated where m equals the number of seizures predicted by the model. p is then the statistical significance level to which our model performs better than the random predictor.

4.3.5 Post-processing

Similar to the previous section, the SVM classifier makes a decision on every window, which we found to produce a noisy signal which causes spurious false alarms.

FPR has an adverse effect on the patient's overall QoL, since high FPRs would keep them in a constant state of alarm and undermining the convenience that such a seizure prediction system would provide them in the first place. To reduce the FPR, it is desirable to devise a method of estimating the true state of the patient based on the cumulative successive outputs of the SVM. Other studies use methods such as threshold crossing or k-of-n.

Threshold crossing, equation 4.3, makes use of the output of the CNN's softmax layer, $q(t)$, and it is used in [42].

$$w(t) = \begin{cases} q(0) & t = 0 \\ \alpha q(t) + (1-\alpha) w(t-1) & t > 0 \end{cases} \tag{4.3}$$

where α is a convex weighting acting as a smoothing parameter. $q(t)$ can be computed from equation 4.4, where

$$q(t) = P\left(c_r \middle| x, \theta\right) = \frac{P\left(x, \theta \middle| c_r\right) P\left(c_r\right)}{\sum_{j=1}^{k} P\left(x, \theta \middle| c_j\right) P\left(c_j\right)} \tag{4.4}$$

where $P\left(c_r | x, \theta\right)$ is the probability that the observation belongs in class r, $P\left(x, \theta | c_r\right)$ is the conditional probability of the observation given the class r, $P\left(c_r\right)$ is the prior class probability, and $\sum_{j=1}^{k} P\left(x, \theta | c_j\right) P\left(c_j\right)$ is the sum of the conditional probabilities and prior class probabilities over all possible classes [74]. Finally, a threshold can be set for $w(t)$ to issue an alarm once it is crossed. This method attempts to make some use of temporal context via the smoothing parameter.

The k-of-n method from [41] simply only issues an alarm if k of the last n windows were predicted to belong to the preictal class. Their results suggest $k = 8$ and $n = 10$ are good choices for a window length of 30 seconds.

We make use of a Kalman filter on the SVM classifications, as in [75]. We implement a second order discrete time Kalman filter which estimates the true state of the patient, x_k, based on noisy outputs of the SVM, y_k, while assuming the true state rate of change is stable with smooth transitions and the measurement noise is much larger than the process noise. This filter can be described by the state space model in section 3.4.2, where the relation $\sigma_v^2 \gg \sigma_w^2$ can be tuned for the desired level of filtering.

4.4 Results

We can evaluate the model trained by the method outlined above on the test set to analyze its performance in predicting seizures. For the purposes of the results listed here, we used a cross-validation approach, which is typical in such applications. Each iteration, one seizure was used for evaluation and all others were used for training. The number of iterations is equal to the number of seizures. Therefore, training and evaluation were accomplished several times for each patient, but each seizure is only used once for evaluation. Further, we repeat this process twice, to ensure consistency between non-deterministic training sessions and to be consistent with [41]. Results for model performance are given in the table below separated by individual patient. Mean and standard deviation between runs is reported for both sensitivity (%) and FPR (/h).

Table 4.1. Summary of results for Boston Children's Hospital dataset.

Pat. ID	**# Seizures**	**Interictal Hrs**	**Sens. [41]**	**FPR [41]**	**Sens. (repr.)**	**FPR (repr.)**	**Sens. (ours)**	**FPR (ours)**	***p***
1	7	17	85.7±0	0.24±0	85.7±0	0.18±0	100±0	0.21±0.04	<0.001
2	3	22.9	33.3±0	0±0	66.6±0	0.09±0.06	66.6±0	0.02±0.03	<0.001
3	6	21.9	100±0	0.18±0	83.3±0	0.14±0	83.3±0	0.11±0.10	<0.001
5	5	13	80±20	0.19±0.03	100±0	0.15±0	80±0	0.19±0.05	<0.001
9	4	12.3	50±0	0.12±0.12	75±0	0±0	100±0	0.04±0.06	<0.001
10	6	11.1	33.3±0	0±0	83.3±0	0.18±0	83.3±0	0±0	<0.001
13	5	14	80±0	0.14±0	60±0	0.21±0.04	60±0	0.07±0.10	<0.001
14	5	5	80±0	0.40±0	100±0	0.40±0	80±0	0.20±0	<0.001
18	6	23	100±0	0.28±0.02	83.3±0	0.22±0.05	91.6±8.35	0.24±0.03	0.002
19	3	24.9	100±0	0±0	33.3±0	0±0	66.6±0	0±0	<0.001
20	5	20	100±0	0.25±0.05	60±20	0.30±0	80±0	0.23±0.04	<0.001
21	4	20.9	100±0	0.23±0.09	100±0	0.24±0.14	100±0	0.17±0.03	<0.001
23	5	3	100±0	0.33±0	80±20	0.66±0	80±0	0.17±0.24	<0.001
Total	64	209	81.2±1.5	0.16±0	77.7±3.0	0.21±0.02	82.4±0.6	0.13±0.06	

Three sets of results are shown. First is the published results in [41], followed by the results we obtained when attempting to reproduce their method and the results obtained by our proposed method outlined in this section.

As shown, our method improves overall sensitivity and FPR of the predictor. While the improvements are marginal from the reported results in [41], our reproduction of their method never reached those presented. The improvement over the reproduction is much better, which may be a better comparison of the two methods. Overall, sensitivity increased by 4.7% and FPR was reduced by 0.08. The increase in sensitivity is likely explained by the inclusion of temporal context features, and the decrease in false alarm rate is likely due to filtering out spurious preictal classifications which might have otherwise raised a warning. Figure 4.7 below shows one such example taken from an hour of interictal data from patient 19. The end result is a

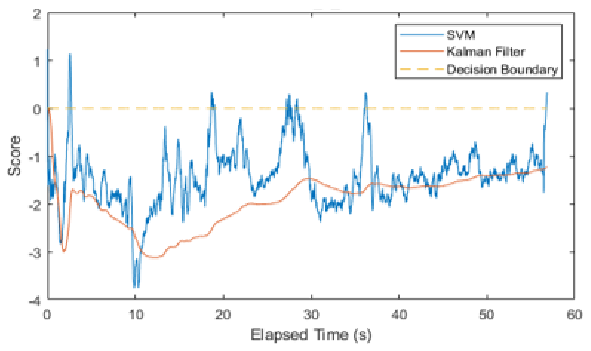


Figure 4.7. Interictal sample from patient 19. Notice the avoided false warnings from spurious preictal classifications.

better seizure predictor and a higher quality of life for the patients.

5. IDENTIFICATION AND MITIGATION OF BYZANTINE AGENTS IN MULTI-AGENT SYSTEMS USING BLOCKCHAIN

5.1 Introduction

In designing distributed networks, especially those which include aerospace vehicles performing sensitive missions or operating over populated areas, it is particularly important to keep security in mind. The advantages of using a distributed network of low cost agents also come with the possibility of failure or even malicious hijacking of agents due to their often minimalist design. In UAV network applications like military surveillance and even industrial package delivery, motivation to disrupt the network is easily found. Additionally, though, even less sensitive applications may experience innocent power or component failures on some nodes. Ideally, networks are designed for resiliency in these cases, so the network's intended goals and behavior are not compromised in these situations.

Potential services provided by UAV networks include package delivery, factory automation, air traffic management, on-demand sensing, transportation, and search and rescue [6–13]. Threats from adversaries and failures might result in the denial of these services, network malfunction, and even leaks of confidential information and user data. Naturally, system designers must address these problems to ensure mission success.

These arbitrary threats, both malicious and naive, can be modeled as Byzantine faults framed by the Byzantine generals problem. First described by [76], the problem describes a faulty process which disseminates conflicting information to other processes. Here, Byzantine faults pose a particular threat to the network's ability to reach a consensus, which is vital for many distributed algorithms which make the network provide any useful function [77]. The distributed consensus problem in the presence of Byzantine faults is often allegorically compared to a group of Byzantine generals surrounding an enemy city. They must decide together whether to attack or withdraw, and some of the generals may be traitors. In order to be successful, all of the loyal generals must decide to attack or withdraw together, otherwise the army will be decimated. Therefore, a solution to the Byzantine generals problem must ensure 1) all loyal generals agree on the same option and 2) the option agreed upon did not explicitly originate from a non-loyal source (i.e. the group does not take the action desired only by the traitors) [77].

Resilience to this type of attack is known as Byzantine fault tolerance. Indeed, several methods of achieving Byzantine fault tolerance have been proposed, since the subject has been studied for over thirty years [78]. However, almost all of these methods require a central authority for process verification, constrain the proportion of dishonest agents to less than a minority, or do not scale well with the size of the network. For instance, the original algorithm proposed by [76] requires a communication cost of $O(n^{m+1})$, where n is the number of network nodes and m is the number of Byzantine agents. Blockchain technology was established to achieve Byzantine fault tolerance for decentralized cryptocurrency markets which have very similar requirements to the UAV networks considered here: decentralized, scalable, and efficient. In fact, Byzantine fault tolerance has since been identified as the most important attribute of blockchain protocols [79]. It could potentially serve as a method of achieving Byzantine fault tolerance in the applications considered here, and it even has the potential to benefit them in other areas like coordination and autonomy once applied as well [5].

5.1.1 Blockchain Byzantine Fault Tolerance

Blockchain achieves Byzantine fault tolerance through the use of a distributed ledger which records and verifies transactions or processes via majority approval, thereby revealing any traitorous ones [79]. Effectively, a blockchain protocol removes the necessity of trust between agents in a distributed network. Blockchains use consensus algorithms to choose a single agent who will produce the next block. Records of processes are grouped together by this agent into blocks which are cryptographically sealed and linked to the previous ones. Each agent stores a local copy of the blockchain which it updates with its neighbors to distribute/receive new blocks. Therefore, only verified blocks are exchanged between agents rather than raw process information. This allows detection of Byzantine faults because logical inconsistencies are recorded on the blockchain and available to each agent even if they occur at large distances across the graph. The most popular consensus algorithm which produces verified blocks and achieves Byzantine fault tolerance on the blockchain information exchanged locally is known as Proof-of-Work (PoW). This was first proposed by Satoshi Nakamoto for Bitcoin but was later adopted in variation by almost all other protocols, including Ethereum [43]. The PoW algorithm is widely successful because it is both difficult to solve and easy to verify. Therefore, blocks are verified quickly and distributed throughout the network. Additionally, since blocks are linked to each preceding one by unique hashes produced by solution of the PoW, a malicious agent seeking to modify a block would have to re-solve PoW problems for every block which follows the one it wishes to modify. Further, all of this computation must be done before the next latest block is produced and all of the hashes are verified again or it must start over. Because of this, the blockchain remains secure until Byzantine agents hold the majority of the network's computing power, which is much less stringent than other methods for Byzantine fault tolerance.

5.1.2 Blockchain Basics

As mentioned before, the blockchain is simply a recording ledger of information and transactions which is secured by the protocols that restrict nodes' ability to edit it. Interactions with the blockchain are usually referred to as transactions involving tokens (the currency). The blockchain is made up of groups of transactions called blocks, which are linked to each other forming a chain. Blocks consist of headers, which include administrative data like the linking information, a timestamp, and other relevant mining data, as well as transaction and potentially other state data organized into merkle trees for accessibility. Figure 5.1 shows a representation of how blocks are usually arranged. The links between blocks, known as hashes, are

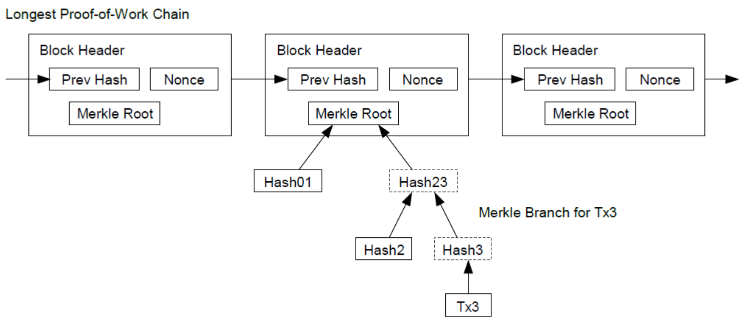


Figure 5.1. The anatomy of a general blockchain [43].

the result of one-way hash algorithms applied to the combination of the previous block's hash, the transaction information included in the block, and an input number called a nonce. The blockchain protocol controls the addition of new transactions to the chain by controlling the process of creating new blocks. Usually, a proof of work algorithm is used to select which participating node will produce the next block. Network nodes known as miners compete to create blocks by finding a suitable nonce to produce a hash in the solution set defined by the protocol. In the proof of work algorithm, the only way to search for a suitable nonce is by brute force, which attaches a computational cost to generating new blocks. The process of attempting to solve the proof of work problems and grouping transactions into candidate blocks is called mining and is shown in Figure 5.2. The target value is set by the blockchain protocol's mining difficulty, and the token reward for successfully mining a new block is known as a coinbase, which incentivizes miner participation. Since each block hash is generated using the hash of the previous block, the blocks are linked together

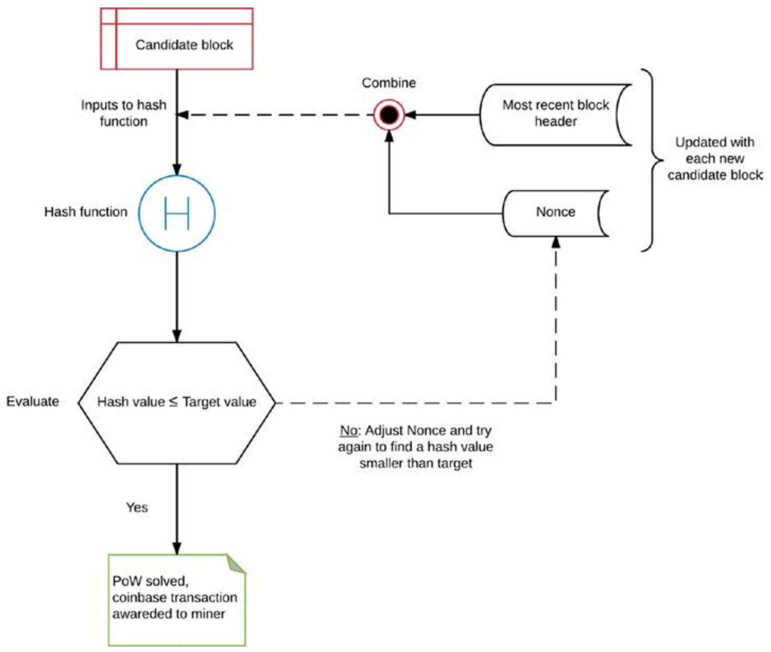

Candidate block
Combine
Inputs to hash function
Most recent block header
Updated with each new candidate block
Hash function
Nonce
Evaluate
Hash value ≤ Target value
No: Adjust Nonce and try again to find a hash value smaller than target
Yes
PoW solved, coinbase transaction awareded to miner

Figure 5.2. Generating a new block with proof of work [79].

such that modifying a block would change the its hash and require calculation of new hashes for all succeeding blocks in the limited time period before a new block is successfully mined. Nakamoto's analysis in [43] shows this provides data security as long as cooperating miners account for at least the majority of processing power in the network.

Since each nodes holds its own version of the blockchain, new blocks must be broadcast when they are created. The successful miner shares the new block with its neighbors, who in-turn share it with theirs to distribute it across the network. It is possible that nodes will be operating off of different versions of the blockchain in sufficiently large or sparse networks. This is called forking, and this is managed by chain length comparisons. The longest chain is traditionally chosen to operate on, so even when nodes produce blocks at the same time, when the versions are rectified when the next block is received. Transactions in the discarded block are returned to the pool for inclusion in the next blocks if they are not already included in other blocks. This process is shown in Figure 5.3. Mining difficulty influences the prevalance

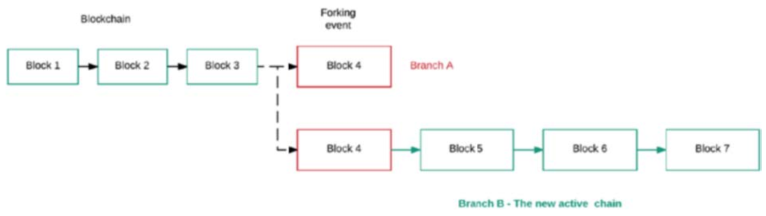


Figure 5.3. Blockchain forking. Two valid blocks exist at position 4, but Branch B is adopted when new blocks are mined [79].

of forking because it controls the block generation rate. When new blocks are created faster than they can be distributed across the network, more possible chain versions exist. Each time a new block is mined and added to the chain, all of the blocks preceeding it are considered validated. For example, a block that is followed by five other blocks has five validations. The canonical chain refers to the section of the blockchain which can be considered stable. That is, it has at least a specified number of validations. The likelihood of losing a block due to forking drops off exponentially with the number of validations that block has received [43]. Therefore, it is useful to operate on information from the canonical chain rather than the most recent blocks.

Blockchains may be implemented publicly, as is the case for Bitcoin and Ethereum, or they may be permissioned or private. Network owners may control who is allowed to participate in the network and, thus, who has access to view or mine on the blockchain. Private blockchains allow companies to enjoy the benefits of distributed record-keeping while still retaining data privacy in-house. Many organizations including Walmart [80], Amazon [6], and Bank of America Merrill Lynch [81] have used this approach to experiment with utilizing blockchain technology in their processes. Additionally, consortiums of similar entities might operate a blockchain together. This acts as a semi-private network, where companies might mutually benefit from provably transparent coordination while retaining privacy from the public. Likewise, data on the blockchain may also be public or private. Traditional key encryption, for example, can be used to ensure transaction records are kept but only relevant stakeholders can access them. This is especially useful for applications involving personal information like medical records or personnel files.

Though the blockchain concept originated with Bitcoin, its implementation there left much to be desired by many who sought to leverage blockchain technology in other applications. While originally created for use only as a cryptocurrency transaction ledger, demand rose for a blockchain architecture which expanded the the allowable data types naturally. Ethereum was created out of this demand, and its focus, rather than secure cryptocurrency exchanges, is being a decentralized, adaptable, public blockchain which supports verifiable scripting. As such, it is a blockchain-based distributed computer which enables users to participate in trustworthy computational interactions. Though not complete, it is a promising first step in an attempt to develop a blockchain-based world computer. Ethereum accomplishes this via the Ethereum Virtual Machine (EVM). The EVM is a Turing-complete virtual processer capable of executing scripts and loading data. As Dhillon et. al write, the "EVM allows any user on the network to execute arbitrary code in a trustless environment where the outcome is fully deterministic and the execution can be guaranteed" [79]. Effectively, this transforms the blockchain from a simple ledger to a computational platform capable of complete isolation of executable code. Each node operates an EVM, which allows nodes to pass inputs to the verified code at the appropriate location on the blockchain and receive the result of the executed functions. Transactions are synonymous with function calls to the code on the blockchain.

Ethereum also consists of the cryptocurrency Ether which, similar to Bitcoin, is used to incentivize miner participation and facilitate the exchange of value in the network. There is also a small Ether cost attached with computation on the blockchain, called gas, which acts as a spam-prevention mechanism. Operations cost gas in the Ethereum network to dissuade code with long run times from clogging up the network. The protocol sets a limit for the amount of gas allowed per block as well as the amount of money users would have to spend to execute this code. The executable code exists on the blockchain in the form of smart contracts, which are called by the

EVM when invoked by the user. Smart contracts exist as byte code which is actually executed when invoking transactions are mined. In turn, the mined block containing this transaction is disseminated through the network as usual, but the executable code is also verified by all miners who receive the new block. This is necessary to ensure all nodes are operating on blockchain versions with the same states. When a full node receives a new block, it executes all transactions included in it to verify it and update its blockchain state. Because of this, computation and storage on the blockchain is relatively expensive. Smart contracts can be written in a programming language like Solidity, which then uses a compiler to convert the program into bytecode for deployment on the blockchain. Then, a node can use the EVM to deploy the contract bytecode with a transaction. When this constructor transaction is mined, the code is included on the blockchain. To interact with the smart contract, a new transaction will is sent with the appropriate arguments. Once this transaction is mined, the code is executed and the blockchain state is updated accordingly. Nodes can watch for the anticipated changes and view the execution results. For further information on the mechanics of Bitcoin, Ethereum, and other common blockchain protocols, see [79].

5.2 Implementation

Ethereum is open source and its source code is modifiable. Because of this, along with the highly-customizable smart contract capabilities, it is often used to experiment with blockchain. Additionally, it supports operation of private networks with multiple nodes from a single machine, which is useful for easily simulating distributed networks. Here, the Ethereum protocol was chosen because it is adaptable to the experiments considered and it enjoys thorough documentation and a large support community. Smart contracts allowed for the implementation of the algorithms considered on the blockchain.

A local private network was run on a single computer to simulate a distributed network operating on the desired algorithms. The process outlined below was repeated for each consensus algorithm considered, with the only differences being the code to implement the algorithms. For each algorithm, consensus was simulated traditionally with no blockchain and all cooperating agents, with a blockchain and all cooperating agents, and then both with and without a blockchain in the presence of Byzantine agents to evaluate the blockchain's effects on resilience. The same network topology is used for all cases of the same algorithm in order to provide a valid comparison between the standard and blockchain cases. For simplicity, simulations including the application of a blockchain will be referred to as the blockchain case. Likewise, the simulations without inclusion of a blockchain are referred to as the standard case.

A MATLAB script was written to simulate the network behavior without the application of a blockchain. This is similar to standard simulations of distributed algorithms, and it operates a specified fixed, synchronous network topology for a specified number of time steps to observe convergence behavior. Another MATLAB script was written to function as a driver for the blockchain case. This driver script performs several operations to initialize the blockchain protocol and pass the desired commands to the EVMs for each node. The MATLAB script interacts with the Ethereum client Geth, which is the command line interface for running Ethereum nodes. For these experiments, Geth's source code was modified and then recompiled to establish a constant mining difficulty to eliminate any effects on the simulations. Normally, Geth employs a variable mining difficulty to control the block generation rate, but this could introduce variance between simulation runs. At each run, the driver uses Geth to initialize a blockchain from a custom genesis block and preallocates 100 Ether into each node's account. This amount is arbitrary but is sufficient to ensure no agents are unable to execute transactions due to a lack of Ether. Next, the individual nodes are initialized and connected as peers in accordance with the simulated network connectivity. This allows nodes to exchange blockchain versions. Next, all nodes begin mining new blocks and one node sends a transaction to deploy the smart contract on the blockchain. The smart contract contains several functions to implement the desired algorithms and is written using the Solidity compiler before the simulation runs. The main functions the nodes interact with are **registerAgent**, which provides the smart contract with the nodes' public keys and initial states, and **applyUpdate**, which executes the update algorithm and returns the agents' next states. All nodes listen for *events* which are triggered when certain transactions are successfully mined and included on the blockchain. When the smart contract deployment transaction is mined, all nodes receive an event notifying them of its location on the blockchain. They will use this address to interact with the smart contract, and, hence, the update algorithm.

Once the nodes receive the smart contract's address, each one sends a signed transaction to the **registerAgent** function with its initial state as an argument. This function records the agent's public key and initial state on the blockchain and returns the block number and block hash where these values are recorded to the agent once the transaction is mined via an event. When all agents have been registered on the blockchain, the blockchain state matches the initial network state in the standard case and the simulation run can proceed. At each time step, each agent sends a transaction to the **applyUpdate** function to receive its next state. The input arguments to this function include the agent's current state and the block number and block hash this state is based on. The **applyUpdate** function checks the agent's current state and block information against the record on the blockchain. The block hash is checked to ensure the agent is operating off of the correct blockchain version and provides the location of the relevant data on the chain. If the block information matches the blockchain's recorded information, the argument state is compared to the recorded state. If this matches, the update algorithm is applied and the new state and corresponding block number and hash are returned to the agent via an event as well as recorded on the blockchain. Locality is enforced since each agent's blockchain version only has the current states of its neighbors it has exchanged chain versions with, whereas the recorded states of non-neighbors is always outdated. If the state does not match, the agent is considered Byzantine, since it holds a state which does not agree with the output of the algorithm at the last time step. When this happens, the smart contract blacklists this agent and ignores it's state values from consideration in the subsequent time steps. This effectively removes the node from the network, since its state will no longer influence the state updates of its neighbors. When a Byzantine agent is identified, it is with 100% certainty. This prevents the Byzantine agents from affecting the other agents once their behavior is identified, which requires one time step. The update algorithm is secured to the level of the blockchain protocol, since its code and execution exist on the blockchain itself. In the same way, the blacklist is also secured to the level of the blockchain protocol and is both immutable and unalterable.

5.3 Unconstrained Consensus

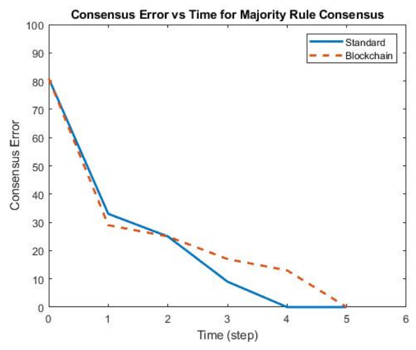
Here, unconstrained consensus refers to a few selected consensus algorithms which are simple yet find applications in multi-agent systems. These algorithms are intended to represent the lowest level of complexity in multi-agent system dynamics and control in order to analyze their utility from a security/resiliency standpoint.

5.3.1 Majority Rule Consensus

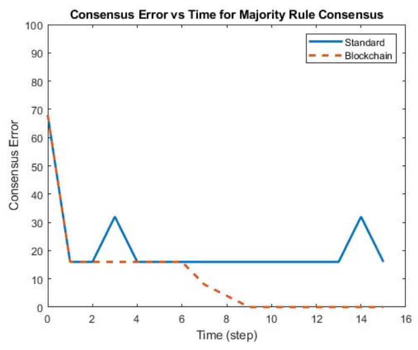
Majority rule consensus is frequently applied in social dynamics models and modeling of natural scientific processes. As previously noted, consensus by majority rule is not guaranteed in every case but depends on network parameters like size, sparsity, and the state value space. Network size and sparsity determine the size of each agent's neighborhood. The state value space determines the possible state values each agent could hold. If the neighborhoods are small and the state value space is large, the likelihood of a state with a clear majority existing in each agent's neighborhood decreases. At its extreme, this situation would cause random states to be adopted at every time step if $x_j(t)$ where $j \in \mathcal{N}_i$ contains no true mode.

Still, it can be shown that the application of a blockchain improves a network's ability to reach a consensus under a majority rule update in the presence of Byzantine agents. Here, we consider Byzantine agents as those which do not update their state in accordance with majority rule. Instead, they hold their state values in order to simulate innocent malfunction Byzantine fault or malicious Byzantine attack seeking to disseminate their state value.

Figure 5.4(a) below shows convergence of an example network consisting of ten non-Byzantine agents under the majority rule update both with and without the application of a blockchain. Convergence is achieved in both the standard distributed consensus case and when a blockchain is applied under the same conditions. Similar convergence behavior is observed, but some slight differences occur due to difference state values selected when an agent's set of neighbor states happens to be multimodal. Since the agent randomly chooses one of the modal states, consensus error is affected when differing states are chosen. Next, two agents of the same network are selected as Byzantine while the initial conditions remain unchanged. Figure 5.4(b) shows convergence behavior both with and without application of a blockchain. The con-



(a) No Byzantine agents.



(b) Two Byzantine agents.

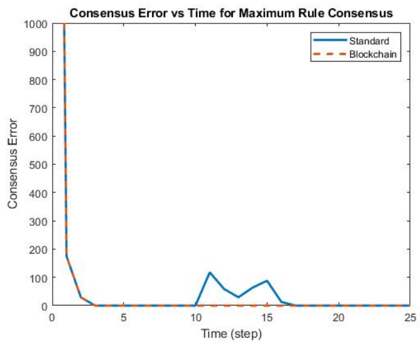
Figure 5.4. Consensus error for ten agents under majority rule.

sensus error history indicates the states of the non-Byzantine agents fail to converge. Only ten time steps are shown for concision, but the final error shown here remains constant even after thousands of time steps. However, when a blockchain is applied under the same conditions, convergence is achieved in a very similar fashion to the non-Byzantine case. Additionally, the convergence value is unaffected by the presence of the Byzantine agents in the blockchain case. This simulation suggests resilience to Byzantine agents is improved by application of a blockchain under majority rule consensus.

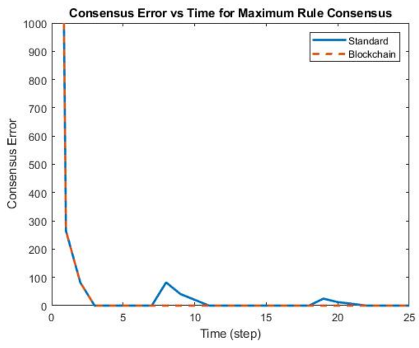
5.3.2 Maximum Rule Consensus

Unlike majority rule, convergence is guaranteed in maximum rule consensus (without inclusion of Byzantine agents), as shown in [82]. Additionally, the state value space is usually increased to real numbers here rather than integers. Maximum rule consensus is often applied when some sort of quality estimate is associated with a measurement or process. Networks seek to reach a consensus of the highest quality estimate for state adoption. Here, agents update their state by selecting the maximum of their set of neighbor states. This proves to be a challenge in combating the effects of Byzantine agents. Because of the time step required for proper identification, states from Byzantine agents can still affect the rest of the network if other agents adopted this state during this lag time.

Figure 5.5(a) below shows convergence of the same ten agent example network both with and without the application of a blockchain under the maximum rule update. Since their is no randomness involved in this update, the convergence behavior should exactly match if initialized the same, which is observed. Under maximum rule, consensus is achieved among cooperating agents in the presence of Byzantine ones both with and without Blockchain. This is due to the fact that the random behavior of Byzantine agents will not have an affect on cooperating ones unless they hold a higher state value, so usually their randomness is ignored. The error occurring once a consensus is reached is caused by Byzantine agents happening to take a state which is greater than the previously converged-upon maximum. This only happens in the standard case, since the blockchain method has already identified and



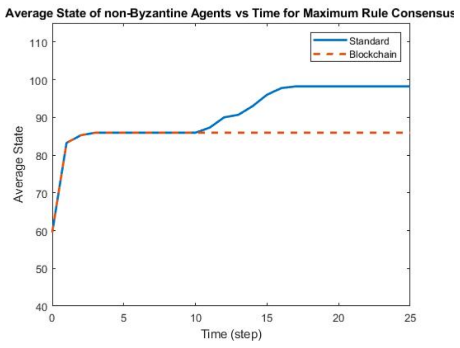
(a) Initial max in Byzantine.



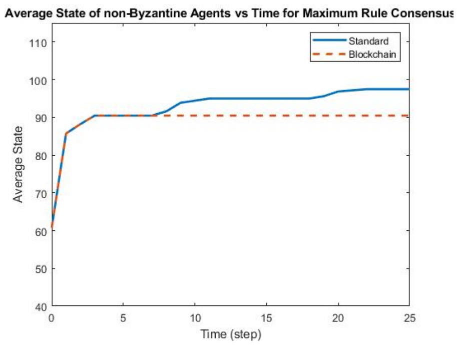
(b) Initial max in non-Byzantine.

Figure 5.5. Consensus error for ten agents under maximum rule.

blacklisted the Byzantine agents by this point. Unless a Byzantine agent holds and disseminates its maximum state before the blockchain can identify it (within the first two time steps), it will not be able to affect the final agreement value. Therefore, although consensus is reached in both cases, the Byzantine agents are much more likely to affect the final agreement value in the standard case. The following plots of the mean state value of cooperating agents vs time illustrate this effect. In the



(a) Initial max in Byzantine.



(b) Initial max in non-Byzantine.

Figure 5.6. Mean state value for ten agents under maximum rule.

blockchain case of Figure 5.6(a), the mean state value decreases after the first time step due to the blockchain's identification of the Byzantine agents and their subsequent blacklisting. However, since one held the true maximum value, its state was already disseminated and adopted by non-Byzantine agents before it was detected who then further distributed it. Convergence under the maximum value update is unaffected by the presence of Byzantine agents as long as they do not isolate non-Byzantine agents from the rest of the network. Further, inclusion of a blockchain of this form does not prevent the network from converging on a state with Byzantine origin. However, the Byzantine agents are correctly identified and recorded on the blockchain, so backtracking and re-execution would allow the state originating from Byzantine agents to be ignored at the expense of a restart.

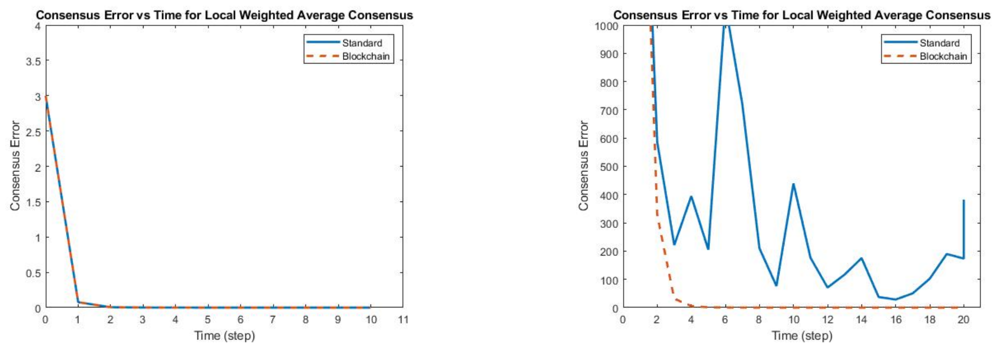
5.3.3 Local Average Consensus

The most widely applied algorithm considered yet, local average consensus has found use in flocking, multivehicle coordination, and sensor networks. For this simulation, the weight matrix, w_{ij}, was chosen to achieve a local direct average update. Specifically,

$$w_{ij} = \begin{cases} \frac{1}{d_i} & j \in \mathcal{N}_\rangle \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

where d_i is the number of neighbors of agent i. At each time step, each agent takes the average of its neighbors' states. This is a simple case of average consensus, but it is still widely used. Choices for w_{ij} enable designers to tweak the convergence value such as Metropolis weights for convergence to the initial global average [83] or Corless weights for convergence to a desired convex combination of the initial states [84].

Similar to the maximum value consensus scenario, for fixed initial conditions, local average consensus behavior should be the same for both the standard and blockchain cases when no Byzantine agents are included. Figure 5.7(a) below validates this conclusion. When two of the agents are Byzantine, however, the standard case fails to converge, as shown in Figure 5.7(b). When a blockchain is applied, convergence is achieved in a similar fashion to the case when no Byzantine agents are included. This is perhaps the clearest example of mitigation of Byzantine effects using blockchain.



(a) No Byzantine agents. (b) Two Byzantine agents.

Figure 5.7. Consensus error for ten agents under local direct average rule.

This simulation suggests resilience to Byzantine faults is improved with the application of a blockchain in networks under the local average update.

5.4 Distributed Solution of Systems of Linear Equations

The solution of linear algebraic equations is one of the most essential capabilities for mathematical problems encountered in science and engineering. In many situations, finding solutions in a distributed manner is advantageous for efficiency, memory, or redundancy reasons. The parallel processing community has pursued this ability for many years because the deconstruction of a large system of linear equations into smaller ones on parallel processors allows faster and more accurate solutions than a direct approach [85, 86]. Further, a distributed method is necessary in some applications due to the physical separation between processors onboard robots or sensors, particularly in swarms [87], sensor networks [15, 88], and some filtering applications [89, 90].

We are interested in a network of agents as previously described, each with their own processing capacity, finding the collective solution to a system of linear equations of which each knows only a part. Specifically, each agent i has a state vector $x_i(t)$ with values $\mathbb{R}^n$ and knowledge of the pair of real-valued matrices $A_i^{n_i \times n}$ and $b_i^{n_i \times 1}$.

Agents communicate locally to drive their state vectors to the solution of the linear equation $Ax = b$ where $A = \begin{bmatrix} A_1' & A_2' & \cdots & A_m' \end{bmatrix}_{n \times \bar{n}}'$, $b = \begin{bmatrix} b_1' & b_2' & \cdots & b_m' \end{bmatrix}_{1 \times \bar{n}}'$, and $\bar{n} = \sum_{i=1}^{m} n_i$. Mou et al. proposed an elegant algorithm to achieve this based on the "agreement principle" [90]. The algorithm forced agents to satisfy their own private equations while iteratively working toward a consensus: the solution to the entire system of equations. This algorithm is

$$x_i(t+1) = x_i(t) - P_i \left(x_i(t) - \frac{1}{d_i} \sum_{j \in \mathcal{N}_i} x_j(t) \right), \qquad j \in \mathcal{N}_i \tag{5.2}$$

where P_i is the projection matrix to the kernel of A_i and d_i denotes the cardinality of $\mathcal{N}_i$. An initialization step requires that $x_i(0)$ satisfies $A_i x_i(0) = b_i$. The projection matrix, P_i ensures each iteration remains in i's local solution space. Therefore, $x_i(0)$ must be initialized in the local solution space. Convergence is guaranteed from the proof in [90] when no Byzantine agents are present. This algorithm was later improved upon by Wang et al. to avoid the initialization step and enable convergence to solutions closest to a particular point by adding another term [91]. This improved algorithm, henceforth referred to as the Distributed Algorithm for Linear Equations (DALE) will be used here and is given as

$$x_i(t+1) = x_i(t) - P_i \left(x_i(t) - \frac{1}{d_i(t)} \sum_{j \in \mathcal{N}_j(t)} x_j(t) \right) - \bar{A}_i' \left(\bar{A}_i \bar{A}_i' \right)^{-1} \left(\bar{A}_i x_i(t) - \bar{b}_i \right) \tag{5.3}$$

where $[\bar{A}_i \ \bar{b}_i]$ denotes a submatrix of $[A_i \ b_i]$ such that ker $A_i =$ ker $\bar{A}_i$ and $\bar{A}_i \bar{A}_i'$ is nonsingular.

Here, Byzantine agents will be modeled as those which do not change their state in accordance with the algorithm above. Instead, they hold their state values in order to simulate innocent malfunction Byzantine fault or malicious Byzantine attack seeking to disseminate their state value. In the blockchain approach, the DALE algorithm exists on the chain in the form of a smart contract. At each time step, agents interact with this smart contract to receive their new state in accordance with DALE. The smart contract records this interaction, as well as the result of the update algorithm.

When agents interact with the smart contract again at the next time step, their current state is compared to the result of the update algorithm at the last step. This allows detection of Byzantine agents which do not hold states in accordance with the DALE algorithm. When these Byzantine agents are detected, they are blacklisted and their states are not considered in future time steps when agents interact with the smart contract to receive their state update. When a Byzantine agent is identified, it is with 100% certainty. This prevents the Byzantine agents from affecting the other agents once their behavior is identified, which requires one time step. The update algorithm is secured to the level of the blockchain protocol, since its code and execution exist on the blockchain itself. In the same way, the blacklist is also secured to the level of the blockchain protocol and is both immutable and unalterable.

In the presence of Byzantine agents, it is expected that a solution to the entire system of equations will be unattainable, since agents hold their own private equations and those known only to the Byzantine agents would only be satisfied by chance. However, it is expected that state convergence will be improved in the presence of Byzantine agents when a blockchain is applied, which would drive the states toward an accurate solution of the system of equations known by the non-Byzantine agents. Two error measures will be used for analysis. First, the solution error shown below measures the difference between the network's states and the solution to the system of linear equations for all non-Byzantine agents.

$$S(t) = \sum_{i=1}^{m} ||x_i(t) - x^*||^2 \tag{5.4}$$

where x^* is the true solution to the system of linear equations. $S(t) = 0$ if and only if all $x_i(t) = x^*$. Second, the consensus error shown below measures the difference between all non-Byzantine agents' states.

$$V(t) = \sum_{(i,j)\in\mathcal{E}} ||x_i(t) - x_j(t)||^2 \tag{5.5}$$

Similarly, $V(t)$ is positive semi-definite and is equal to zero only when a consensus is reached.

The numerical simulation below uses a network of three agents with connectivity corresponding to the adjacency matrix

$$A_{\mathbb{G}} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} \tag{5.6}$$

attempting to solve the system of linear equations shown below.

$$A = \begin{bmatrix} 1 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 1 \end{bmatrix}, \; b = \begin{bmatrix} -6 \\ -9 \\ 15 \end{bmatrix} \tag{5.7}$$

which has the unique solution

$$x^* = \begin{bmatrix} 0 \\ 6 \\ 9 \end{bmatrix} \tag{5.8}$$

When no Byzantine agents are present, this system of equations is solved identically by the standard and blockchain methods, as shown below. Since $S(t)$ goes to zero, all state vectors have converged to x^*.

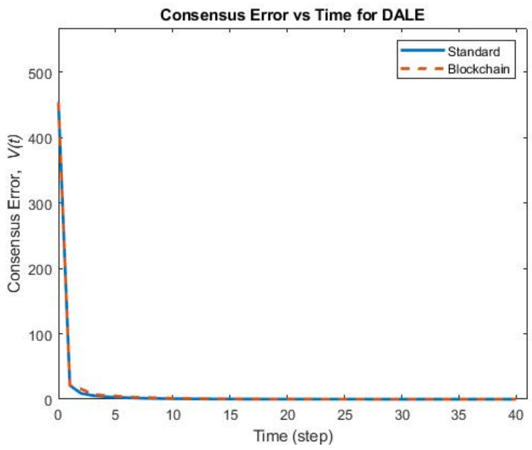


Figure 5.8. Solution error for example simulation without inclusion of Byzantine agents.

However, when one agent is Byzantine, the standard method fails to converge at all, let alone converge near x^*. The errors shown here only consider the non-Byzantine agents ($\{\bar{i} \in i : \bar{i}$ is non-Byzantine$\}$).

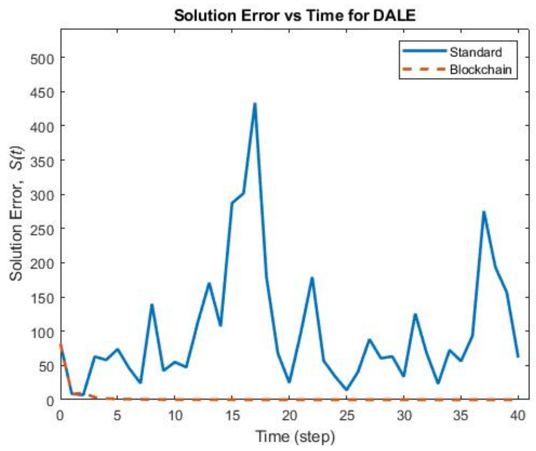


Figure 5.9. Solution error for example simulation with a single Byzantine agent.

As expected, the blockchain method also fails to reach $S(t) = 0$, but it does exhibit different behavior than the standard case. $S(t)$ indicates that the blockchain method increases solution accuracy, presumably due to the improvement of state convergence expected. Since the states of the non-Byzantine agents converge, their private equations are satisfied simultaneously, whereas the failure of state convergence in the standard case only ensures each agent's private equations are satisfied independently by arbitrary state vectors. Examination of the consensus error supports this finding, indicating state convergence is achieved in the blockchain approach as opposed to the standard approach.

When no Byzantine agents are present, both methods converge to the correct solution. The slight trace differences are only due to the number precision difference for the blockchain case, since Ethereum does not support fixed point operations. When Byzantine agents are considered, the standard case has no hope of converging, let alone to the correct solution. In the blockchain case, convergence is achieved, but since some information from the problem is lost (only the Byzantine agents know

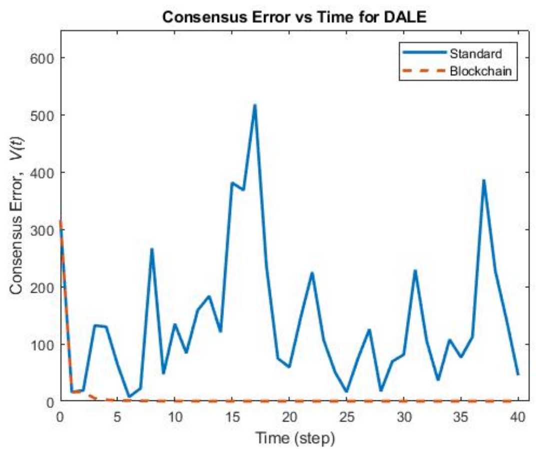
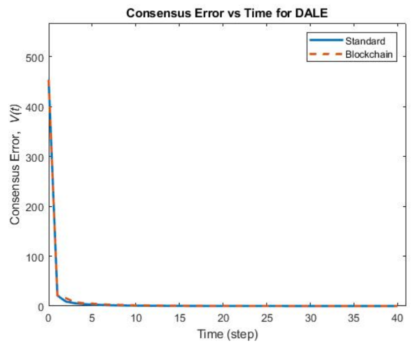
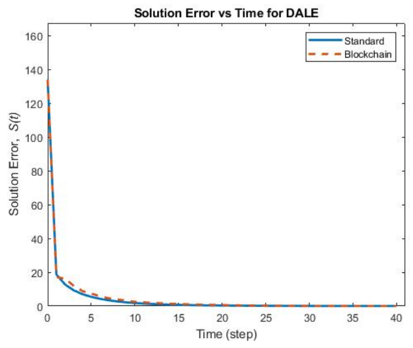


Figure 5.10. Consensus error for example simulation with a single Byzantine agent.



(a) Consensus error.

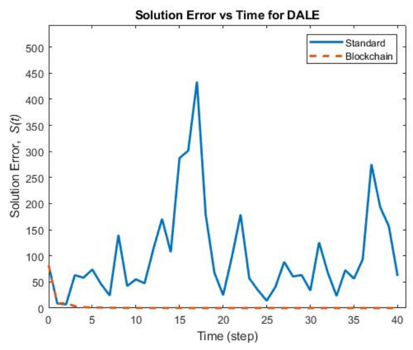


(b) Solution error.

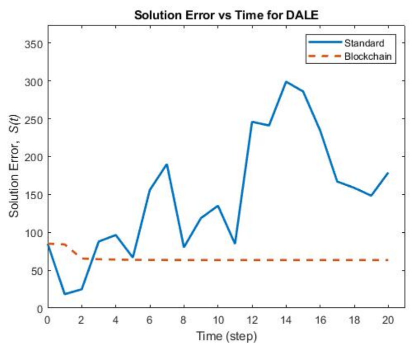
Figure 5.11. Consensus and solution error with no Byzantine agents.

their equation rows), convergence to the correct solution is not guaranteed. Instead, the cooperating agents will converge upon a solution to the problem posed by the remaining known equation rows, which will likely have multiple solutions. The figures below show convergence to the correct solution by chance (when initial values were close to x^*) and a different solution. In both cases, all known equations in the system are satisfied.

This simulation suggests resilience to Byzantine faults is improved with the application of a blockchain in networks under the DALE algorithm. Further, the converged



(a) Convergence to x^*.



(b) Convergence to different solution.

Figure 5.12. Consensus and solution error with a single Byzantine agent.

solution will satisfy the private equations of all non-Byzantine agents, unlike the standard case in which all private equations are only satisfied independently.

5.5 Constrained Distributed Optimization

Yet another desirable ability for multiagent systems is distributed constrained optimization. In these problems, the goal is to optimize a global objective function which is the sum of local agent objective functions, subject to a constraint set given by the intersection of the local agent constraint sets. Particularly, each agent i holds a state vector, $x_i(t)$, which is its estimate of the solution to the optimization problem. Each agent knows only its own local objective function f_i, its own constraint set X_i, and the states of its neighbors at the last time step $x_j(t)$. We seek to utilize only local interaction for the agents to cooperatively solve the constrained optimization problem

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} f_i(x) \\ \text{subject to} \quad & x \in \bigcap_{i=1}^{m} X_i \end{aligned} \tag{5.9}$$

where $f_i : \mathbb{R}^n \to \mathbb{R}$ is a convex objective function and $X_i \subseteq \mathbb{R}^n$ is a closed convex constraint set. Let the optimal value of this problem be denoted as f^* which occurs at x^*.

The ability to solve optimization problems in a distributed manner opens the door for extremely capable distributed networks. This process could be applied to wireless sensor networks and even robot swarms to accomplish global objectives in an optimal manner. Still, the distributed nature of the problem allows calculation to take place over many agents with potentially decreased processing capabilities. Applications of distributed constrained optimization include traffic control [92], environment parameter estimation [93], smart grid resource allocation [94], and even artificial intelligence [95].

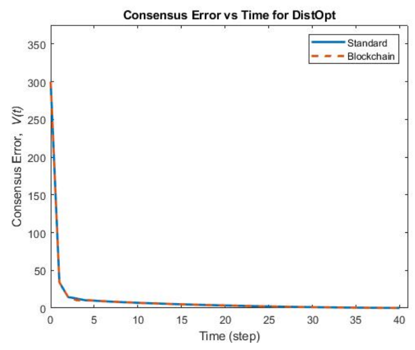
Here, we will consider the work of Nedić et al. and use a distributed subgradient method for solving the distributed optimization problem above. Similar to DALE discussed previously, this algorithm is rooted in the agreement principle and consensus is used as a mechanism for distributing computations among agents. Each agent updates its state by combining the states of its neighbors, taking a subgradient step to minimize its objective function, f_i, and finally by projecting on its constraint set X_i [96]. Each agent starts with an initial estimate of the solution which satisfies its own constraints, $x_i(0) \in X_i$. The distributed update algorithm is

$$v_i(t) = \sum_{j=1}^{m} a_i^j(t) x_j(t) \tag{5.10}$$

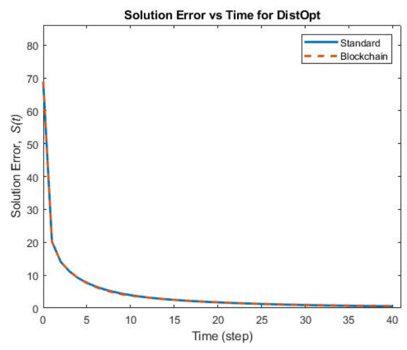
$$x_i(t+1) = P_{X_i}\left[v_i(t) - \alpha_t g_i(t)\right] \tag{5.11}$$

where the scalar $a_i^j(t)$ is the nonnegative weight of the edge from agent i to agent j, α_t is the step size, $g_i(t)$ is the vector subgradient of $f_i(x)$ at $x = v_i(t)$. In [96], it is shown that this algorithm converges without the presence of Byzantine agents for the case of uniform weights $a_i^j(t) = (1/m)$ for all i and j, i.e., $\mathbb{G}$ is fully connected.

When no Byzantine agents are present, both methods converge to the correct solution. When Byzantine agents are included, again the standard case has no hope of convergence or convergence to a solution. The blockchain case converges to the optimal value of the known objective function, which is different from the actual objective function since the local objective functions of the Byzantine agents are not known to the system. Still, the converged upon value is very close to x^* since the



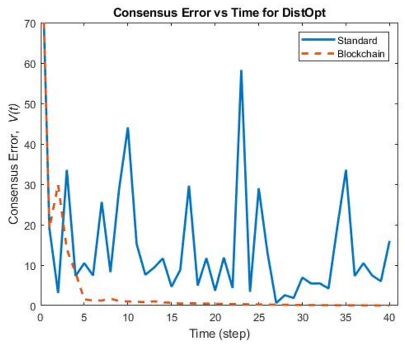
(a) Consensus error.



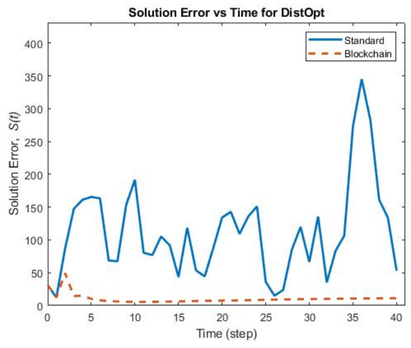
(b) Solution error.

Figure 5.13. Consensus and solution error with no Byzantine agents.

global objective function is the sum of all local ones. In the blockchain case, both consensus and solution error increase in the time step that the Byzantine agents are identified. This is expected from the lag before they are blacklisted, since Byzantine behavior must first occur to be recognized on the blockchain. They quickly recover once this happens, however.



(a) Consensus error.



(b) Solution error.

Figure 5.14. Consensus and solution error in the presence of Byzantine agents.

5.6 Associated Cost

Even though blockchain has been hailed as the way of the future for many fields, its mechanisms impose a significant computational cost. In the applications considered here, each agent would act as a miner working to add transactions and new blocks to the chain. The traditional method which keeps this process secure, proof of work, relies on computational capacity being expended by each miner. Therefore, by simply seeking to solve the proof of work problem posed by each block, additional computations are required compared to the standard method. The difficulty of these problems can be regulated by the protocol's source code. Reducing the difficulty would decrease the amount of computational expense spent mining blocks but would increase the block generation rate. An increased block generation rate could result in a higher risk of agents holding forked chain versions. Additionally, blocks might be created before there are any transactions to include in them, wasting storage space on each node which still must store its header. Normally, blockchain protocols adjust mining difficulty to achieve a desired block generation rate. On the main Ethereum blockchain, this is about one block every 15 seconds. A balance must be found between mining difficulty and block generation rate to achieve a reasonable number of transactions per block. The expected rate of transactions depends on the particular application and implementation. More computationally efficient alternatives to the proof of work algorithm are in development. Ethereum plans to implement the proof of stake algorithm which replaces solving a math problem by brute force with establishing stake-holding measures to determine which miner creates the next block. Proof of stake drastically reduces the computational cost of mining on the blockchain, and it is expected to be implemented in the very near future [97].

Perhaps the greatest computational cost of using a blockchain for these applications is the replication of operations performed in each transaction. The Ethereum protocol was designed to be a distributed computer focused solely on trust rather than speed. By allowing executable code to exist on the blockchain, computations can be performed and information shared between completely trustless nodes. Effectively, it is a protocol which ensures that processes are carried-out in accordance with predefined rules among any agents. It replaces the need for institutional, administrative, or other sources of trust between agents. Here, rule enforcement in the form of properly updating state values in accordance with the prescribed algorithm is mechanical and transparent as opposed to private calculations done on each agent's processor. To accomplish this, the Ethereum protocol requires that transactions are validated on every node when chain information is updated. This means each agent will perform the calculations of all preceding transactions under this protocol. In distributed systems with limited processing power, this does not scale well with network size. The computational complexity of this verification process for each individual agent is $O(n)$, where n is the number of nodes on the network, since each agent must validate each **applyUpdate** transaction. Still, this is a framework to ensure all agents play by the same rules in a completely trustless network and could be especially useful in applications which favor trust and accountability over speed. This is only true for the Ethereum protocol, however, so custom protocols could be developed which require less validation calculations on a private network. This would most likely be the best way to apply these techniques in a real-world scenario. While inefficient due to the validation burden, Ethereum is used here as a proof-of-concept. Additionally, research into implementing the same concept in protocols which do not require verification on every node and drastically reduce the imposed computational cost is ongoing [79].

Additionally, there are communication and storage costs associated with the blockchain case, since agents no longer simply exchange state information but entire block information, which includes data overhead in the form of headers and raw transaction information like public keys, gas costs, and previous states. On the main Ethereum blockchain, blocks sizes are about 25 kilobytes and the entire chain length is about 8.5 million blocks as of July 2019. On a private network for the applications considered here, however, both the block size and the chain length would likely be just a fraction of this since the main Ethereum network supports transactions for millions of users and decentralized applications. In the simulations presented here, the average block size is 500 bytes for the local average unconstrained consensus problems. The chain length depends mostly on the number of time steps required to achieve consensus, which will vary widely across applications based on factors like the spread of initial values and network topology. At each time step, agents do not exchange entire versions of the blockchain including all recorded data, but rather only new blocks are exchanged and deconflicted. Therefore, the communication cost is the message size of the new block information and its recorded transactions, and the storage cost is the entire preceding portion of the blockchain. For each agent, these also scale with $O(n)$, where n is the number of agents, since each agent requires a transaction at each time step. The continual growth of data on the blockchain is known as bloat, and it also afflicts the main Ethereum chain as well on a much larger scale. Since the nodes in a multi-agent system are likely to have limited memory capacity, storing the entire blockchain might be infeasible. To retain the blockchain's integrity, however, block's must be linked to all of those preceding it by their proof of work. In order to limit storage requirements while also assuring the blockchain's integrity, only essential block information could be retained as the blocks age. For blocks whose recorded data (like outdated state information or gas costs) is no longer relevant, only the headers which link blocks together could be stored instead, which would reduce the memory requirements for individual agents.

5.7 Conclusion

While applying blockchain technology to UAV swarms offers many potential benefits, there are difficulties associated with using blockchain in UAV networks. [98] identifies a number of technical challenges related to blockchain, some of which may be of particular importance when attempting to use it in resource-limited networks like UAV swarms. In particular, latency seems like the biggest technical problem to face, particularly in very large networks. In a blockchain, latency is the difference in time between execution of a transaction and its acceptance into the blockchain. [5] notes that Bitcoin blocks take about ten minutes to be processed before they are added to the chain. Many swarm applications (e.g. flocking) require very quick information exchange between network agents to accomplish. Excessive latency could result in very poor flocking ability and even collisions. Much of the time it takes to process a Bitcoin block is spent calculating the block's proof of work. Alternatives to computationally difficult problems for proof of work equivalents include concepts such as "proof of stake" [99]. Another possible solution to this problem is affiliation trust, where agents belonging to a specific organizations might be trusted more easily than others, but this might have potential security concerns. [5] calls for innovative research in the security versus speed tradeoff area. Another challenge associated with using blockchain technology is size and throughput. Especially in large networks, the blockchain might grow to the point where all of its information cannot fit in the agents' memory. The Bitcoin community refers to this problem as "bloat" [5, 100]. This is especially important in UAV networks, where each agent's storage size and processing power is limited compared to networks of computers.

REFERENCES

[1] Federal Aviation Administration. FAA National Forecast FY 2019-2039 Full Forecast Document and Tables. page 105, 2019.

[2] Eric. Bonabeau, Marco. Dorigo, and Guy. Theraulaz. *Swarm intelligence : from natural to artificial isystems*. Oxford University Press, 1999.

[3] Sifat Momen. Ant-inspired decentralized task allocation strategy in groups of mobile agents. In *Procedia Computer Science*, volume 20, pages 169–176, 2013.

[4] Joanne H Walker and Myra S Wilson. Task allocation for robots using inspiration from hormones. *Adaptive Behavior*, 19(3):208–224, 2011.

[5] Eduardo Castelló Ferrer. The blockchain: A new framework for robotic swarm systems. *Advances in Intelligent Systems and Computing*, 2019.

[6] Amazon.com Inc. Amazon.com: Prime Air, 2016.

[7] Matt MacFarland. UPS drivers may tag team deliveries with drones, 2017.

[8] Tiago M Fernández-Caramés, Oscar Blanco-Novoa, Manuel Suárez-Albela, and Paula Fraga-Lamas. An UAV and Blockchain-based System for Industry 4.0 Inventory and Traceability Applications. In *International Electronic Conference on Sensors and Applications*, volume 15, 2018.

[9] Hsun Chao, Apoorv Maheshwari, Varun Sudarsanan, Shashank Tamaskar, and Daniel A. DeLaurentis. UAV Traffic Information Exchange Network. In *2018 Aviation Technology, Integration, and Operations Conference*, 2018.

[10] Airobotics. Automated Industrial Drones, 2018.

[11] Uber Air — Uber Elevate.

[12] Jürgen Scherer, Bernhard Rinner, Saeed Yahyanejad, Samira Hayat, Evsen Yanmaz, Torsten Andre, Asif Khan, Vladimir Vukadinovic, Christian Bettstetter, and Hermann Hellwagner. An Autonomous Multi-UAV System for Search and Rescue. In *Proceedings of the First Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use - DroNet '15*, pages 33–38, 2015.

[13] Vincenzo Lomonaco, Angelo Trotta, Marta Ziosi, Juan de Dios, Yanez Avila, and Natalia Diaz-Rodriguez. Intelligent Drone Swarm for Search and Rescue Operations at Sea. *Artificial Intelligence*.

[14] Iván García-Magariño, Raquel Lacuesta, Muttukrishnan Rajarajan, and Jaime Lloret. Security in networks of unmanned aerial vehicles for surveillance with an agent-based approach inspired by the principles of blockchain. *Ad Hoc Networks*, 86:72–82, apr 2019.

[15] Lin Xiao, Stephen Boyd, and Sanjay Lall. A scheme for robust distributed sensor fusion based on average consensus. pages 63–70, 2005.

[16] Brian D.O. Anderson, Changbin Yu, Soura Dasgupta, and A. Stephen Morse. Control of a three-coleader formation in the plane. *Systems and Control Letters*, 56(9-10):573–578, 2007.

[17] Naomi Ehrich Leonard, Derek A Paley, Francois Lekien, Rodolphe Sepulchre, David M Fratantoni, and Russ E Davis. Collective motion, sensor networks, and ocean sampling. *Proceedings of the IEEE*, 95(1):48–74, 2007.

[18] Brian D.O. Anderson, Changbin Yu, Bans Fidan, and Julien M. Hendrickx. Rigid Graph Control Architectures for Autonomous Formations. *IEEE Control Systems*, 2008.

[19] Shaoshuai Mou, Mohamed Ali Belabbas, A Stephen Morse, Zhiyong Sun, and Brian D.O. Anderson. Undirected rigid formations are problematic. *IEEE Transactions on Automatic Control*, 61(10):2821–2836, 2016.

[20] Ming Cao, Changbin Yu, and Brian D.O. Anderson. Formation control using range-only measurements. *Automatica*, 47(4):776–781, apr 2011.

[21] Shaoshuai Mou, Fenghua He, and Silun Zhang. Formation Tracking Based on Target Points. *IFAC-PapersOnLine*, 48(28):921–926, 2015.

[22] Frank L. Lewis, Draguna L. Vrabie, and Vassilis L. Syrmos. *Optimal Control: Third Edition*. John Wiley & Sons, Inc., Hoboken, NJ, USA, jan 2012.

[23] S. Bhasin, R. Kamalapurkar, M. Johnson, K. G. Vamvoudakis, F. L. Lewis, and W. E. Dixon. A novel actor-critic-identifier architecture for approximate optimal control of uncertain nonlinear systems. *Automatica*, 2013.

[24] Luis Rodolfo Garcia Carrillo and Kyriakos G. Vamvoudakis. Deep-Learning Tracking for Autonomous Flying Systems Under Adversarial Inputs. *IEEE Transactions on Aerospace and Electronic Systems*, PP(8):1–1, 2019.

[25] Girish Chowdhary and Eric Johnson. Concurrent learning for convergence in adaptive control without persistency of excitation. In *Proceedings of the IEEE Conference on Decision and Control*, pages 3674–3679, 2010.

[26] Ben Nassi, Asaf Shabtai, Ryusuke Masuoka, and Yuval Elovici. SoK - Security and Privacy in the Age of Drones: Threats, Challenges, Solution Mechanisms, and Scientific Gaps. pages 1–17, 2019.

[27] Zhiwei Feng, Nan Guan, Mingsong Lv, Weichen Liu, Qingxu Deng, Xue Liu, and Wang Yi. Efficient drone hijacking detection using onboard motion sensors. In *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, pages 1414–1419, 2017.

[28] Zhiwei Feng, Nan Guan, Mingsong Lv, Weichen Liu, Qingxu Deng, Xue Liu, and Wang Yi. An efficient UAV hijacking detection method using onboard inertial measurement unit. *ACM Transactions on Embedded Computing Systems*, 17(6):96, 2019.

[29] Hongjun Choi, Wen-Chuan Lee, Yousra Aafer, Fan Fei, Zhan Tu, Xiangyu Zhang, Dongyan Xu, and Xinyan Deng. Detecting Attacks Against Robotic Vehicles: A bv Control Invariant Approach. In *CCS*, 2018.

[30] Samy Kamkar - SkyJack: autonomous drone hacking.

[31] Khurum Nazir Junejo and Jonathan Goh. Behaviour-Based Attack Detection and Classification in Cyber Physical Systems Using Machine Learning. (Ml):34–43, 2016.

[32] Savio Sciancalepore, Omar Adel Ibrahim, Gabriele Oligeri, and Roberto Di Pietro. Picking a Needle in a Haystack: Detecting Drones via Network Traffic Analysis. 2019.

[33] Waylon D Scheller. Detecting Drones Using Machine Learning. *ProQuest Dissertations and Theses*, page 35, 2017.

[34] Home - Mars Parachute.

[35] Drone Parachute, Multicopter Parachute, UAV, RC Aircraft Recovery — Fruity Chutes!

[36] Epilepsy Foundation of America. About Epilepsy: The Basics — Epilepsy Foundation, 2014.

[37] Mayo Clinic. Epilepsy - Symptoms and causes - Mayo Clinic, 2017.

[38] About Epilepsy — CDC, 2018.

[39] Florian Mormann, Ralph G Andrzejak, Christian E Elger, and Klaus Lehnertz. Seizure prediction: The long and winding road, 2007.

[40] M Winterhalder, T Maiwald, H U Voss, R Aschenbrenner-Scheibe, J Timmer, and A Schulze-Bonhage. The seizure prediction characteristics: A general framework to assess and compare seizure prediction methods. *Epilepsy and Behavior*, 4(3):318–325, 2003.

[41] Nhan Duy Truong, Anh Duy Nguyen, Levin Kuhlmann, Mohammad Reza Bonyadi, Jiawei Yang, Samuel Ippolito, and Omid Kavehei. Convolutional neural networks for seizure prediction using intracranial and scalp electroencephalogram. *Neural Networks*, 105:104–111, sep 2018.

[42] Haidar Khan, Lara Marcuse, Madeline Fields, Kalina Swann, and Bulent Yener. Focal Onset Seizure Prediction Using Convolutional Networks. *IEEE TRANSACTIONS ON BIOMEDICAL ENGINEERING*, 65(9), 2018.

[43] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Technical report, 2008.

[44] Stephen Majercik. Initial Experiments in Using Communication Swarms to Improve the Performance of Swarm Systems. In Fernando A. Kuipers and Poul E. Heegaard, editors, *Self-Organizing Systems: 6th IFIP TC 6 International Workshop*, Lecture Notes in Computer Science, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[45] Aryo Jamshidpey and Mohsen Afsharchi. Task Allocation in Robotic Swarms: Explicit Communication Based Approaches. In *Advances in artificial intelligence: 28th canadian conference on artificial intelligence*, volume 9091, 2015.

[46] Levent Bayindir. A review of swarm robotics tasks. *Neurocomputing*, 172:292–321, 2016.

[47] I A Zikratov, I S Lebedev, E V Kuzmich, and A V Gurtov. Securing Swarm Intellect Robots with a Police Office Model. Technical report, 2014.

[48] Alan G Millard, Jon Timmis, and Alan F T Winfield. Towards Exogenous Fault Detection in Swarm Robotic Systems. In *Conference Towards Autonomous Robotic Systems*, 2014.

[49] Fiona Higgins, Allan Tomlinson, and Keith M. Martin. Survey on security challenges for swarm robotics. In *Proceedings of the 5th International Conference on Autonomic and Autonomous Systems, ICAS 2009*, pages 307–312, 2009.

[50] Vishal Sharma, Ilsun You, and Gökhan Kul. Socializing Drones for Inter-Service Operability in Ultra-Dense Wireless Networks using Blockchain. In *Proceedings of the 2017 International Workshop on Managing Insider Security Threats - MIST '17*, pages 81–84, 2017.

[51] Alexander Kuzmin and Evgeny Znak. Blockchain-base structures for a secure and operate network of semi-autonomous Unmanned Aerial Vehicles. In *Proceedings of the 2018 IEEE International Conference on Service Operations and Logistics, and Informatics, SOLI 2018*, pages 32–37, 2018.

[52] Laurent Reynaud and Isabelle Guérin-Lassous. Physics-based swarm intelligence for disaster relief communications. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9724, pages 93–107, 2016.

[53] John Arquilla and David Ronfeldt. Swarming and the Future of Conflict. Technical report, National Defense Research Institute, 2005.

[54] Simon Thiel, Dagmar Häbe, and Micha Block. Co-operative Robot Teams in a Hospital Environment. In *IEEE International Conference on Intelligent Computing and Intelligent Systems*, 2009.

[55] Julien M. Hendrickx, Brian D.O. Anderson, Jean Charles Delvenne, and Vincent D. Blondel. Directed graphs for the analysis of rigidity and persistence in autonomous agent systems. *International Journal of Robust and Nonlinear Control*, 17(10-11):960–981, jul 2007.

[56] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 1990.

[57] Girish Chowdhary, Tansel Yucelen, Maximillian Mühlegg, and Eric N. Johnson. Concurrent learning adaptive control of linear systems with exponentially convergent bounds. *International Journal of Adaptive Control and Signal Processing*, 27(4):280–301, apr 2013.

[58] Ilias Giechaskiel and Kasper B. Rasmussen. Taxonomy and Challenges of Out-of-Band Signal Injection Attacks and Defenses. *IEEE Communications Surveys & Tutorials*, pages 1–1, 2019.

[59] Devaprakash Muniraj and Mazen Farhood. Detection and mitigation of actuator attacks on small unmanned aircraft systems. *Control Engineering Practice*, 83:188–202, feb 2019.

[60] Jayaprakash Selvaraj, David Ware, Gökçen Yılmaz Dayanikli, Ryan M Gerdes, Neelam Prabhu Gaunkar, and Mani Mina. Electromagnetic induction attacks against embedded systems. In *ASIACCS 2018 - Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security*, volume 18, pages 499–510, 2018.

[61] Drew Davidson, Hao Wu, Robert Jellinek, Thomas Ristenpart, and Vikas Singh. Controlling UAVs with Sensor Input Spoofing Attacks. Technical report.

[62] Todd E. Humphreys, Brent M. Ledvina, Mark L. Psiaki, Brady W. O'Hanlon, and Paul M. Kintner. Assessing the spoofing threat: Development of a portable gps civilian spoofer. In *21st International Technical Meeting of the Satellite Division of the Institute of Navigation, ION GNSS 2008*, 2008.

[63] Denis Foo Kune, John Backes, Shane S. Clark, Daniel Kramer, Matthew Reynolds, Kevin Fu, Yongdae Kim, and Wenyuan Xu. Ghost talk: Mitigating EMI signal injection attacks against analog sensors. In *Proceedings - IEEE Symposium on Security and Privacy*, 2013.

[64] Yunmok Son, Hocheol Shin, Dongkwan Kim, Youngseok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[65] Timothy Trippel, Ofir Weisse, Wenyuan Xu, Peter Honeyman, and Kevin Fu. WALNUT: Waging Doubt on the Integrity of MEMS Accelerometers with Acoustic Injection Attacks. In *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017*, 2017.

[66] Ang Cui, Michael Costello, and Salvatore J Stolfo. When Firmware Modifications Attack : A Case Study of Embedded Exploitation. *20th Annual Network Distributed System Security Symposium*, 2013.

[67] Sergei Skorobogatov. Local heating attacks on flash memory devices. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2009*, 2009.

[68] Orrin Devinsky. Sudden, Unexpected Death in Epilepsy. *New England Journal of Medicine*, 365(19):1801–1811, nov 2011.

[69] David J. Thurman, Dale C. Hesdorffer, and Jacqueline A. French. Sudden unexpected death in epilepsy: Assessing the public health burden. *Epilepsia*, 55(10):1479–1485, oct 2014.

[70] Ali Shoeb. Application of machine learning to epileptic seizure onset detection and treatment. *Diss. Massachusetts Institute of Technology*, 2009.

[71] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Technical report.

[72] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. Technical report.

[73] Björn Schelter, Matthias Winterhalder, Thomas Maiwald, Armin Brandt, Ariane Schad, Andreas Schulze-Bonhage, and Jens Timmer. Testing statistical significance of multivariate time series analysis techniques for epileptic seizure prediction. *Chaos*, 2006.

[74] Christopher M. Bishop. *Machine Learning and Pattern Recoginiton*. Springer Science+Business Media, 2007.

[75] Yun Park, Lan Luo, Keshab K. Parhi, and Theoden Netoff. Seizure prediction with spectral power of EEG using cost-sensitive support vector machines. *Epilepsia*, 2011.

[76] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[77] Wenbing Zhao. *Building Dependable Distributed Systems*, volume 9781118549. Scrivener Publishing, Beverly, 2014.

[78] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[79] Vikram Dhillon, David Metcalf, and Max Hooper. *Blockchain Enabled Applications*. Apress, New York, 2017.

[80] Walmart Explores Blockchain for Delivery Drones — Fortune.

[81] Microsoft and Bank of America Merrill Lynch collaborate to transform trade finance transacting with Azure Blockchain as a Service, 2016.

[82] Reza Olfati Saber and Richard M Murray. Consensus Protocols for Networks of Dynamic Agents. In *Proceedings of the 2003 American Control Conference*, 2003.

[83] Lin Xiao and Stephen Boyd. Fast linear iterations for distributed averaging. *Systems and Control Letters*, 53(1):65–78, 2004.

[84] Leonardo Coduti and Martin Corless. A decentralized algorithm for assigning the weighting parameters in a general synchronous consensus network. In *51st IEEE Conference on Decision and Control*, 2012.

[85] K. Koç, A. Güvenç, and B. Bakkalo Ğ Lu. Exact solution of linear equations on distributed-memory multiprocessors. *Parallel Algorithms and Applications*, 3(1-2):135–143, 1994.

[86] Christer Andersson. Solving Linear Equations on Parallel Distributed Menory Architectures by Extrapolation. Technical report, 1997.

[87] Jian Yang, Xin Wang, and Peter Bauer. Line and v-shape formation based distributed processing for robotic swarms. *Sensors (Switzerland)*, 18(8), 2018.

[88] Soummya Kar, José M.F. Moura, and Kavita Ramanan. Distributed parameter estimation in sensor networks: Nonlinear observation models and imperfect communication. *IEEE Transactions on Information Theory*, 58(6):3575–3605, 2012.

[89] Usman A Khan and José M F Moura. Distributed Kalman filters in sensor networks: Bipartite fusion graphs. In *Proc. Workshop Statist. Signal Process*, pages 700–704, 2007.

[90] Shaoshuai Mou, Ji Liu, and A Stephen Morse. A distributed algorithm for solving a linear algebraic equation. *IEEE Transactions on Automatic Control*, 60(11):2863–2878, 2015.

[91] Xuan Wang, Shaoshuai Mou, and Dengfeng Sun. Improvement of a Distributed Algorithm for Solving Linear Equations. *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS*, 64(4):3113, 2017.

[92] Robert Junges and A.L.C. Bazzan. Evaluating the performance of DCOP algorithms in a real world, dynamic problem. *AAMAS*, (Aamas):599–606, 2008.

[93] Victor Lesser, Charles Otriz, and Milinde Tambe. *Distributed Sensor Networks: A Multiagent Perspective.* Springer Science+Business Media, New York, 2003.

[94] Tsung Hui Chang, Angelia Nedić, and Anna Scaglione. Distributed constrained optimization by consensus-based primal-dual perturbation method. *IEEE Transactions on Automatic Control*, 59(6):1524–1538, 2014.

[95] Allan R. Leite, Fabrício Enembreck, and Jean Paul A. Barthès. Distributed Constraint Optimization Problems: Review and perspectives, sep 2014.

[96] Angelia Nedic, Asuman Ozdaglar, and Pablo A Parrilo. Constrained consensus and optimization in multi-agent networks. *IEEE Transactions on Automatic Control*, 55(4):922–938, 2010.

[97] Vitalik Buterin and Virgil Griffith. Casper the Friendly Finality Gadget. Technical report.

[98] Melanie. Swan. *Blockchain: Blueprint for an Economy.* O'Reilly Media, Sebastopol, first edit edition, 2015.

[99] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10401 LNCS, pages 357–388. Springer, Cham, 2017.

[100] Andrew Wagner. Ensuring Network Scalibility: How to Fight Blockchain Bloat — Bitcoin Magazine, 2014.