SAND2021-1623C

Moving Target Defense for Space Systems

Chris Jenkins, Eric Vugrin, Indu Manickam, Nicholas Troutman, Jacob Hazelbaker, Sarah Krakowiak
Sandia National Laboratories
Albuquerque, USA
{ cdjenk, edvugrin, imanick, ntroutm, jshazel, skrakow}@sandia.gov

Josh Maxwell
Qualtrics
Albuquerque, NM
jmaxwell@qualtrics.com

Richard Brown
Tennessee Technological University
Cookeville, Tennessee
Brown, Richard rgbrown42@tntech.edu

***Abstract*—Space systems provide many critical functions to the military, federal agencies, and infrastructure networks. Nation-state adversaries have shown the ability to disrupt critical infrastructure through cyber-attacks targeting systems of networked, embedded computers. Moving target defenses (MTDs) have been proposed as a means for defending various networks and systems against potential cyber-attacks. MTDs differ from many cyber resilience technologies in that they do not necessarily require detection of an attack to mitigate the threat. We devised a MTD algorithm and tested its application to a real-time network. We demonstrated MTD usage with a real-time protocol given constraints not typically found in best-effort networks. Second, we quantified the cyber resilience benefit of MTD given an exfiltration attack by and adversary. For our experiment, we employed MTD which resulted in a reduction of adversarial knowledge by 97%. Even when the adversary can detect when the address changes, there is still a reduction in adversarial knowledge when compared to static addressing schemes. Furthermore, we analyzed the core performance of the algorithm and characterized its unpredictability using nine different statistical metrics. The characterization highlighted the algorithm has good unpredictability characteristics with some opportunity for improvement to produce more randomness.**

Keywords—cyber resilience, moving target defense, intelligent system scrambling, real-time network security, cybersecurity, MIL-STD-1553, machine learning

I. Introduction

Space systems provide many critical functions to the military, federal agencies, and infrastructure networks. Nation-state adversaries have shown the ability to disrupt critical infrastructure through cyber-attacks targeting systems of networked, embedded computers. This knowledge raises concern that space systems could face similar threats, and increasing the resilience of space systems to cyber-attacks is a growing area of research. Many proposed cyber resilience technologies require detection of threats before mitigative actions can be taken. Because reliable detection of cyber threats is still a significant challenge to terrestrial systems, reliance on threat detection for cyber resilience can be a risky strategy.

Moving target defenses (MTDs) have been proposed as a means for defending various networks and systems against potential cyber-attacks. MTDs differ from many cyber resilience technologies in that they do not necessarily require detection of an attack to mitigate the threat. However, little empirical evidence exists to prove that MTDs do improve cyber resilience. Furthermore, MTDs increase operational complexity. For real-time systems, this complexity could potentially result in unacceptable delays, decreased reliability, and other negative impacts.

This paper describes research and development of a MTD algorithm that can be used to defend real-time space systems against potential cyber-attacks. This paper specifically seeks to address the two following research questions:

1) Does the use of the MTD algorithm introduce unacceptable, negative impact on the operations of a selected, space platform?

2) Does the use of the MTD algorithm provide measurable benefits to the cyber resilience of a selected, space platform?

Because of the widespread use of MIL-STD-1553 buses in avionics and space systems [1], we selected a MIL-STD-1553 bus system as our exemplar space platform for conducting our research.

The rest of this paper is as follows: Section II provides background on MTD, the MIL-STD-1553 protocol, and related work. Section III describes the algorithm design, and how we have integrated the algorithm into the MIL-STD-1553 protocol. Section IV provides a unpredictability characterization of the MTD algorithm that assesses how adjustments to MTD algorithm features can affect the ability of an adversary to potentially defeat the MTD. Section V describes reliability and cyber resilience experiment designs, and Section VI contains results from the experiments. Section VII provides our conclusions and future work.

II. Background

A. Moving Target Defense

MTDs create dynamic, seemingly uncertain environments which seek to confuse the attacker and attempt to defeat cyber threats [2]. Attackers attempting to perform network discovery, exfiltrate data, and other malicious activities will observe that key system attributes change in a seemingly chaotic and random manner. In truth, these changes are choreographed among the system components in a manner such that they know when and how the changes will occur.

For example, a webserver may change its IP address every day to a different, randomly selected address to thwart an attacker's attempt to hack the webserver. Another option is to randomize the port that the webserver listens on to respond to

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

requests. Other more complex variations exist as well. To our knowledge, MTD has mainly been applied in traditional information technology (IT) environments [3]-[5], and Chavez et. al's research is one of the few efforts applying MTDs to industrial control systems [6]. Furthermore, most previous studies into MTD effectiveness have been qualitative and survey-based while some attempt to quantify the benefits[7]-[9].

Space systems typically have components and on-vehicle networks that are not IP-based. They utilize real-time protocols which must adhere to determinism, predictability, reliability, and real-time deadlines. Furthermore, some protocols do not have cybersecurity protection mechanisms (e.g., authentication, encryption). MTD provides an opportunity to add cyber resilience in the absence of typical cybersecurity controls. However, the potential impacts of MTD on space systems' operations and resilience have not been previously quantified.

B. MIL-STD-1553

MIL-STD-1553 is a military standard for transmitting data on a reliable data bus [10]. Typical implementations use a dual-redundant bus in failover mode to improve reliability. This bus is typically found in military systems (e.g., weapon systems, aircraft, helicopters) as well in space systems (e.g., space satellites). Though the protocol was invented in the 1970s, it has remained a staple as a command and control network for a variety of systems due to its reliability.

The bus gains much of its reliability by using a dual-redundant bus and a master-slave communication scheme. The protocol specifies a bus A and a bus B where bus B is idle unless bus A detects no traffic. Since only one device uses the bus at a time (i.e., time-division multiplexing), collisions are avoided on the bus—assuming each device on the bus adheres to the protocol.

The MIL-STD-1553 uses three types of devices: a bus controller (BC), up to 31 remote terminals (RTs), and a bus monitor (BM). The BC controls all communication on the bus. The BC sends commands on the bus which instruct the specific RT. The RT may consume or send data based on the command. All other RTs remain silent on the bus until they are instructed by the BC. The BM records all commands and data that are sent on the bus. The BM never sends any data; it is a read-only device and serves the same purpose as a flight recorder on an airplane.

The MIL-STD-1553 protocol has 3 types of data within the protocol: command word (CW), data word (DW), and status word (SW). Each word type is 16 bits (2 bytes) in length. The command word is only sent by the BC and commands each RT. Both the BC and RT send data words based on the CW sent by the BC. Status words are sent only by RTs and inform the BC of correct reception of the CW and DWs (if present). Furthermore, each bit on the bus consumes one microsecond, and the bus operates at 1 MHz Each word consists of a 3 microsecond sync, 16 bits (16 microseconds), and one parity bit (1 microsecond).

The MIL-STD-1553 has two types of messages: non-broadcast (i.e., unicast) and broadcast. As the names imply, non-broadcast messages result in communication between 2 devices while broadcast messages result in communication between a sending device and all receiving devices on the network. These messages are organized into frames. A frame is an organized collection of messages. A network has 2 or 3 types of frames. A minor frame represents the highest frequency in the network and represents the base unit of scheduling. A major consists of one or more minor frames (e.g. 10 minor frames). Some networks utilize a higher frame structure called super frames, which consists of one or more major frames. For example, a minor frame may repeat at a frequency of 50 Hz. Collecting 8 minor frames into a major frame produces a frequency of 6.25 Hz for the major frame. Collecting 4 major frames into a super frame produces a frequency of 1.56 Hz for the super frame.

The MIL-STD-1553 protocol does not include authentication or encryption features. The lack of these protections has raised concerns about risks that systems using this protocol are vulnerable to cyber-attacks.

C. Related Work

Attempts have been made to improve the security of MIL-STD-1553 using a sequence-based anomaly detection methodology. In particular, Stan et al. used 3 different machine learning models to detect unauthorized data transmissions and spoofing attacks [11]. The authors also characterize the bus by learning what messages are sent and the frequency of such messages. As mentioned before, this paper focuses on anomaly detection.

Recent work has examined MTD and other real-time protocols. The controller area network bus or CAN bus has a similar network topology as MIL-STD-1553. The *CAN ID Shuffling Technique* (CIST) utilizes encryption and hashing to produce a sequence-dependent shuffling of CAN IDs on a per-message basis [12]. For each message, up to two keyed-based message authentication codes (i.e., HMACs) and one encryption is needed. The technique requires a variety of different crypto-variables: group keys, one-time keys, seeds, encryption keys, etc. With all the aforementioned keys and algorithms, the authors devised a scheme where the CAN ID can change per message (typically CAN IDs don't change).

Brown et al. suggested a model which uses a seed value and dynamic state called *dynamic address validation array* of DAVA [13]. In their scheme, each engine control unit or ECU invalidates its CAN ID when needed, and the other devices use an algorithm to synchronize the new CAN ID for the respective node. As with CIST the CAN IDs change but without all the state and algorithm complexity.

III. MTD Algorithm and Integration

Use of the MTD algorithm within a real-time protocol requires that certain constraints must be addressed. These key constraints are as follows:

- **Keep underlying protocol** - These protocols have been designed for determinism, predictability, reliability, and real-time operation. By using the existing protocol as is, we seek to avoid introducing effects which detract from the real-time properties of these protocols.
- **Dynamic address generation** - Each node in the network should employ the same MTD algorithm, produce the same address state, and index disjoint addresses as compared to all other nodes in the network.

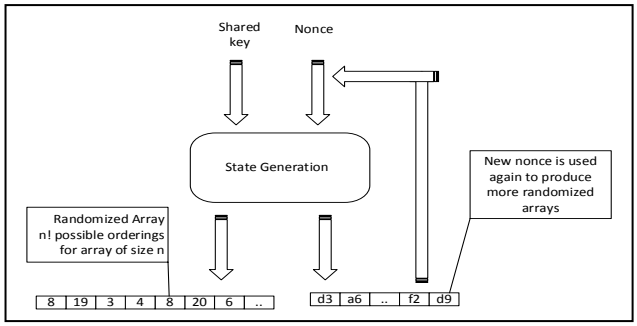


Fig. 1. State generation architecture

- **Synchronization** – All devices on the network must be synced up as the addresses change over time. Each device must understand how and when addresses change. Extended loss of synchronization could have catastrophic consequences for real-time protocols.
- **Authenticity** - The devices on the network should be able to decipher authentic MTD commands from non-authentic MTD commands. Using analog attributes, message authentication codes, message integrity checks, etc., with the MTD command provides an avenue to ensure only valid MTD commands are followed.

Our algorithm assumes only one address will be used at a time. However, nothing precludes nodes from utilizing multiple dynamic addresses concurrently or splitting up a message and sending the message parts to different addresses.

A. MTD Algorithm

At a high level, our MTD algorithm consists of 3 architectural components: 1) a shared key and an ephemeral token (i.e., cryptographic nonce), 2) state generation, and 3) the manner of selecting an address from the state (i.e., indexing).

We assume that the shared key is generated and shared offline (i.e., not sent on the bus) between the participating nodes while the ephemeral token can be sent on the bus. Our algorithm uses a pseudo-random number generator (PRNG) to assist with state generation. In general, PRNGs utilize a variety of underlying algorithms to generate numbers such as: 1) a naïve generator using modular arithmetic (e.g., linear-feedback shift register), 2) a keyed-hash message authentication code (HMAC)-based generator, or 3) a stream/block cipher-based generator. Lastly, the manner of selecting an address (i.e., indexing) can vary. For example, the new address can be selected with a fixed index or the index can move within the state. Below we will describe the architecture's components.

1) Keys

Two keys are needed for the MTD algorithm: shared and ephemeral. The shared key is only shared between nodes on the network participating in MTD. The ephemeral key (i.e., nonce) is shared between all participating nodes but not necessarily kept private from others (i.e., non-participating nodes or adversaries). Both keys are 128-bit in length, and the algorithm can employ the use of larger keys.

2) State Generation

Each participating node uses the state to select the new address. The state generation occurs over a set of rounds as shown in Fig. 1. Each round has 3 parts:

- Random byte generation
- Column (i.e., array) generation
- Permutation

a) Random Byte Generation

Our algorithm uses a PRNG to assist with state generation. The PRNG attempts to generate a non-repeating sequence of bytes (i.e., 8 bits) as shown in Fig. 2. Typically, the PRNG begins its sequence by utilizing a seed value. Different seed values produce different non-repeating sequences. In general, PRNGs utilize a variety of underlying algorithms to generate numbers. For our algorithm, we used a keyed-hash message authentication code (HMAC) based generator.

For MIL-STD-1553 usage, the PRNG uses the shared key and ephemeral key to produce 64 bytes. Given that each byte has a range between 0 – 255, there is a chance that some of the bytes will be duplicates. As more bytes are produced, the chance of a duplicate byte increases.

b) Conditioner

The conditioner constructs addresses from those bytes specific to the protocol being used. Using the 64 bytes from the random byte generation stage, the conditioner:

- Creates an array of 31 random bytes from the 1st 31 bytes of the 64 generated bytes (33 remaining bytes)
- Reduces each byte to a value between 0 and 30
- Removes duplicate bytes and inserts missing address values
- Removes addresses not available for usage on the network (reduces array size)
- Shuffles the array to add more randomness

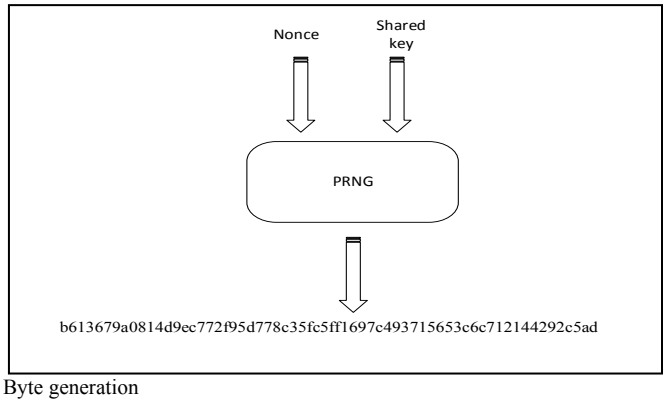


Fig. 2. Byte generation

At the end of these set of steps, the state consists of a single array which is a random ordering of addresses from 0 to 30. Given that there are 31 possible addresses for each cell in the column, there are 31! possible unique columns as shown in Fig. 3.

c) Permutation

The last step of state generation produces a new ephemeral key using these steps:

- Take the last 32 bytes (from the original 64) and split them into two 16-byte halves
- XOR those halves resulting in a 16-byte value
- Using the 32nd byte (the remaining byte from the original 64 bytes) perform left and right circular rotations on alternating set of 4 bytes (4 rotations on the 16-byte value).

This process continues until the required number of arrays (i.e., columns or state arrays) has been produced (maximum of 65536 columns for our work) as shown in Fig. 4

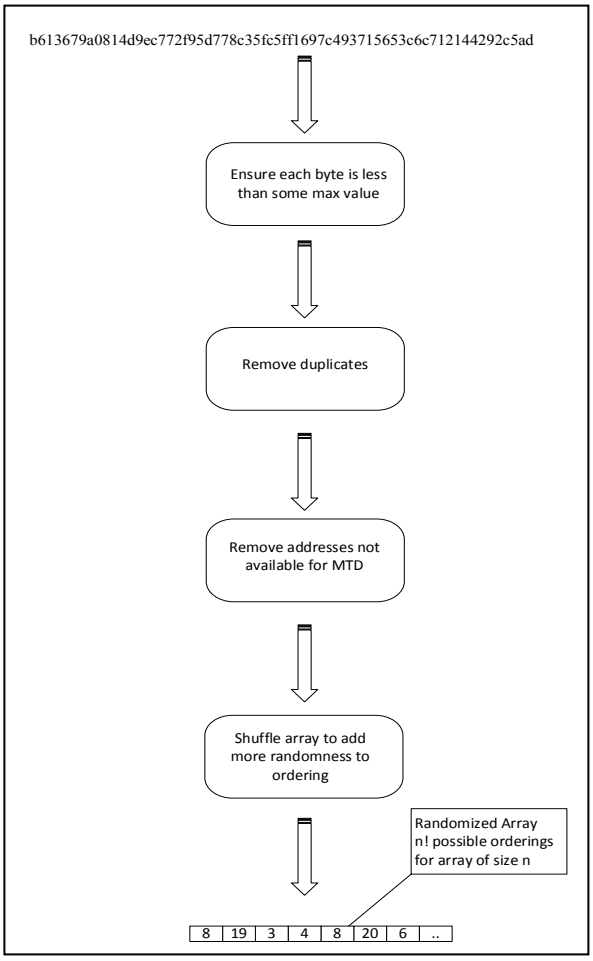


Fig. 3. Column conditioning

3) Indexing

Once we have the MTD state generated, we need a way to select the new address from the state. We use two values to pick the new address: a 16-bit index value and address value. The index value selects the column (i.e., array) and the address value selects the offset into the column. We have 4 different combinations of selecting the new address which we label as static, current, and linear-static, linear-current.

a) Static

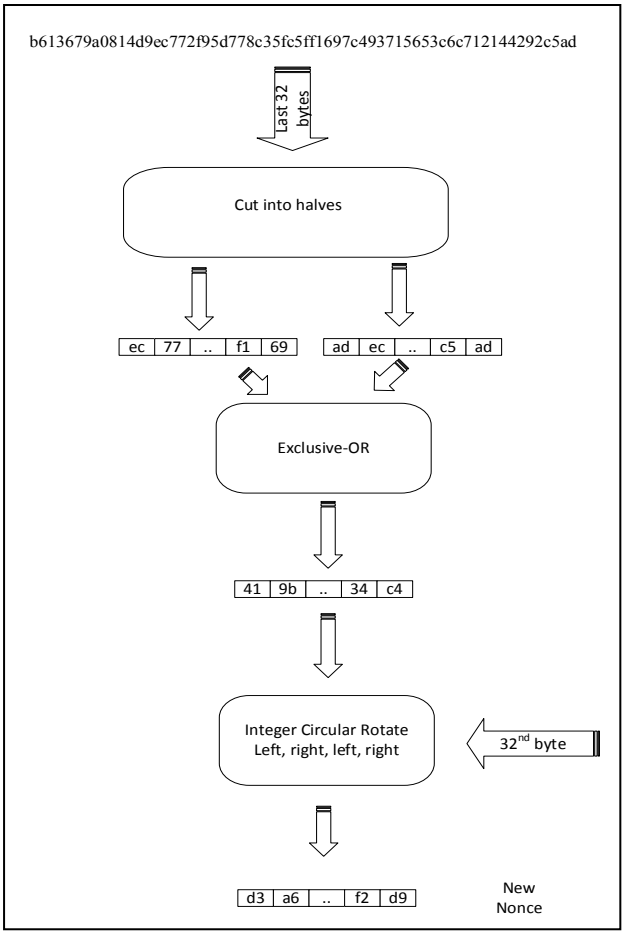


Fig. 4. Permutation

This offsetting mechanism uses the same value for the address value. Typically, this value is set to the non-MTD address of the node (i.e., device) on the network. For example, consider the case illustrated in Fig. 5. Assume that an index value of 0, and a node has a starting address of 5. That node is assigned the address located at offset 5 (zero-based offset value). For this state, the node is assigned a value of 20. If the index value changes to 1, 2, or 3, then the assigned address changes to 19, 1, or 30 respectively. The offset value stays at 5 (the starting address) regardless of the index value (i.e., the selected column).

b) Current

This offsetting mechanism uses the current address of the node for the address value. For example, consider the case illustrated in Fig. 5 and assume a starting address of 3. The first selection of the new address with use an offset value of 3 (zero-based offset value). The new address has a value of 4. If the index value changes to 1, use an offset value of 4 to find the next address to be 5. If the index value then changes to 3, the offset value changes to 5, and the new address will be 30. Therefore, for each index, the same address is not likely to be returned.

c) Linear

This mechanism uses the index value and breaks it up into sub-values. The first ten bits are used as the index (i.e., limits the number of columns to 1024). The remaining six bits are used to

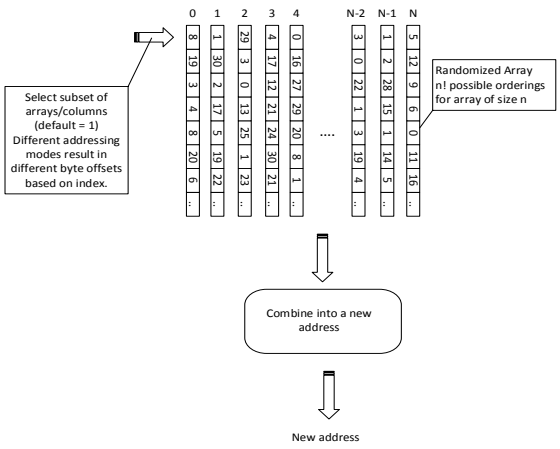


Fig. 5. State matrix

construct the offset based on the linear combination of these values. We use the equation:

$$4a + b + c \bmod 31 = d \qquad (1),$$

where a is a 3-bit unsigned number, b is a 3-bit unsigned number, c is the current (or initial) address, and d is the offset value. The linear combination produces an offset value which is used to select the new address. A breakdown of the 4 combinations are listed in TABLE I.

TABLE I. OFFSET SELECTION TABLE

Offset Selection Mechanism		**Index Interpretation**	
		16-bit unsigned integer	***Linear combination***
Address Used	***Initial address***	Static	Linear-static
	Current address	Current	Linear-current

B. MIL-STD-1553 Integration

In order to use the algorithm with the MIL-STD-1553 protocol, our mechanism must integrate with the existing protocol. Our design uses 4 MTD commands based on the existing MIL-STD-1553 command structure. The four commands are:

- MTD Start – this command sends a 128-bit nonce, the size of the state to generate, offsetting mechanism, and any addresses which are not permitted for use in the state matrix.
- MTD Verify – this command sends a 256-bit keyed-hash message authentication code (HMAC) to the requestor. This enables the host node (e.g., BC) to verify all participating nodes have the same state without leaking information.
- MTD Update – this command sends an index (i.e., indirect) value which participating nodes use to select their new address. In our experiments, the index value selects the state column.
- MTD Stop – this command instructs all participating nodes to stop listening to MTD Update commands. The nodes can either revert to their initial address or stay at their current address. The host node would indicate this choice in the MTD Start command.

The four MTD commands and their respective number of command words and data words are listed in TABLE II. To minimize the impact on the existing frame structure, we attempted to limit the additional words needed to implement our scheme. Based on our scheme, we determined the MTD Update command consumes a majority of the cost of adding MTD to an existing frame structure. For the MTD Update command, we require one command word and one data word which translates into 40 microseconds.

TABLE II. MTD COMMANDS

Command	**CWs**	**DWs**	**Broadcast**	**Time sent**
MTD Start	1	9 - 16	Yes	1
MTD Verify	1	16	No	Number of MTD nodes
MTD Update	1	1	Yes	Every n frames
MTD Stop[1]	1	1	Yes	1

As stated before, these commands could be spoofed. For our design, we suggest using a 256-bit HMAC (16 DWs) when sending any MTD commands. This would increase the amount of time the MTD Update command consumes on the bus. A variety of schemes exist to construct an appropriate message authentication code (MAC) and are outside of the scope of this paper. To keep devices in sync, a MTD Update command could be sent at the end of minor, major, and/or super frames.

IV. UNPREDICTABILITY OF THE ADDRESS ASSIGNMENTS

The MTD algorithm is designed to generate sequences of address updates for each node over time that are complex, avoid repeating patterns, and are difficult for an attacker to predict. The MTD algorithm contains several parameters that can be adjusted to affect the unpredictability of the address sequence. In this section, we analyze how different system parameter settings affect the inherent unpredictability of the MTD-generated sequences. We draw on concepts from information theory to perform this analysis without having to make explicit assumptions about an attacker's capabilities.

A. System Parameters for Evaluation

We examined two system parameters, the offset method and the number of state matrix columns. The specific values tested for this experiment are shown in TABLE III. In addition, we

[1] Technically, the MTD Stop command does not require any data words. However, MIL-STD-1553 doesn't support non-mode code broadcast with zero data words.

examined the effect of updating the state matrix on address unpredictability, as detailed in the next section.

TABLE III. SYSTEM PARAMETERS ANALYZED FOR THE UNPREDICTABILITY STUDY

System Parameter	Values
Offset Method	Static, Current, Linear-S, Linear-C
State Matrix Columns	128, 256, 512
State Matrix Updates	No Updates, 1 Update

B. Experimental Setup

The input data for this experiment is a sequence of addresses generated by the MTD algorithm. This sequence can essentially be treated as a time series, where each timestep corresponds to an address update. For simplicity, we limit our analysis to address assignment for a single node, and leave it as a future effort to ensure that the address assignments are independent across nodes.

In total we generated 3,720 address sequences, and each sequence consists of 4,096 address updates. The following process was used to generate the address sequences:

1. Create 10 state matrices using 10 PRNG seeds.
2. For each state matrix, generate 31 address sequences for each of the 12 unique combinations of offset and number of state matrix columns.
3. To simulate the effect of state matrix updates, create 5 pairs of state matrices at random from the set of 10.
   a) For each pair, extract the first half (2,048 address updates) for all sequences generated using the two state matrices.
   b) Concatenate the two half sequences from each state matrix for each parameter combination. This will simulate the effect of switching state matrices halfway through the sequence while retaining the same system parameters.

Given this data set of address sequences, we used the following process to evaluate unpredictability:

1. Calculate a set of nine unpredictability metrics (detailed in Section C), and average each metric over the 31 address sequences per state matrix and unique parameter setting.
2. Compute the Overall Predictability Metric per state matrix and parameter setting by averaging across all metric values.
3. For each system parameter under analysis, for example the choice of offset method, average across all address sequences that use the same parameter value.

C. Unpredictability Metrics

We use the following set of nine metrics to quantify unpredictability of the MTD-generated address sequences. The metrics quantify entropy, complexity, and turbulence using different criteria. We use the term "unpredictability" to represent the difficulty of predicting the sequencing as measured by the group of these nine metrics. Third party python packages were used to implement the permutation, spectral, sample, and SVD entropy, as well as the Lempel-Ziv complexity [14].

- Shannon's Entropy: Measures the uniformity of address assignments for node x using an empirical estimate of frequency.
- Lempel-Ziv Complexity: Measures complexity based on the compressibility of the time series [15].
- Sample Entropy: Determines the regularity and patterns in the data and how that varies over increasing window sizes [16].
- Permutation Entropy: Measures how subsets of the time series relate to each other based on the ordinal rankings of values within each subset [17].
- Spectral Entropy: Measures the distribution of frequencies in the data by computing the entropy of the power spectral density of the data.
- SVD Entropy: Computes the entropy of the singular values of a matrix built from subsets of the time series.

The remaining 3 metrics were generated using a transition matrix. This matrix is an N x N matrix where N is the number of addresses. The entry in the i^{th} row and j^{th} column represents the empirical probability that a node assigned to address i will subsequently be assigned to node j.

- Transition Entropy: Averages the Shannon's entropy of each row of the transition matrix.
- Transition Uniformity: Averages the distance between each row distribution and a uniform distribution [18].
- Transition Turbulence: Measure the weighted edge density of the transition matrix, with the weights determined by the effective degree of each node [18].

All of the metrics were normalized to the range [0,1], where 1 indicates higher unpredictability. Given that each metric evaluates unpredictability based on a different set of criteria, we also compute an *Overall Unpredictability* score by averaging across the nine metric values. Due to differences in sensitivity, interpretation, and values between the nine metrics, this score should not be treated as an absolute measure of unpredictability. Instead, the score can be used to evaluate the effect of system parameters on unpredictability, by examining trends in whether the score increases or decreases as the system parameters are varied.

D. *Unpredictability Study Results*

The Overall Unpredictability Scores for each system parameter setting are shown in TABLE IV. In addition, we plot the averaged metric scores for each system parameter setting in **Fig. 6**.

TABLE IV. OVERALL UNPREDICTABILITY SCORE FOR SYSTEM PARAMETERS

Number of State Matrix Columns	**Overall Unpredictability Score**
512	***0.8497 ± 0.0306***
256	0.8456 ± 0.0395
128	0.8399 ± 0.0443
Offset Method	**Overall Unpredictability Score**
Linear-C	***0.8797 ± 0.0006***
Linear-S	0.8795 ± 0.0008
Current	0.8183 ± 0.0067
Static	0.8002 ± 0.0155
State Matrix Updates	**Overall Unpredictability Score**
1 State Matrix Update	***0.8482 ± 0.0349***
No Updates	0.8451 ± 0.0389

All scores exceed 0.8. Given a maximum possible score of 1.0, these scores provide confidence in the MTD algorithm. Use of the linear methods results in higher unpredictability scores than those resulting from the current and static approaches. The number of columns and use of state matrix updates does not appear to have a significant effect on the Unpredictability Scores.

Linear methods, in particular Linear-C, produces sequences that have higher unpredictability than the other offset methods. Updating the state matrix at least once as well as increasing the number of state matrix columns does also slightly improve unpredictability.

As shown in **Fig. 6**, the set of unpredictability metrics are either unaffected or similarly affected by the parameter values. In particular the metrics based on the transition matrix, the Transition Turbulence and the Transition Uniformity, are the most sensitive to changes in the system parameters.

This experiment allows us to demonstrate quantitatively that the Linear-C or Linear-S offset methods are preferable and provide a set of tools to evaluate future parameter changes.

V. RELIABILITY AND RESILIENCE EXPERIMENTS

We considered two experiments to evaluate the MTD algorithm. We use the first experiment to assesses whether "the use of the MTD algorithm introduces unacceptable, negative impact on the operations" on MIL-STD-1553 hardware. We use the second experiment to assess whether "use of the MTD algorithm provides measurable benefits to the cyber resilience" of an application running on the MIL-STD-1553 network

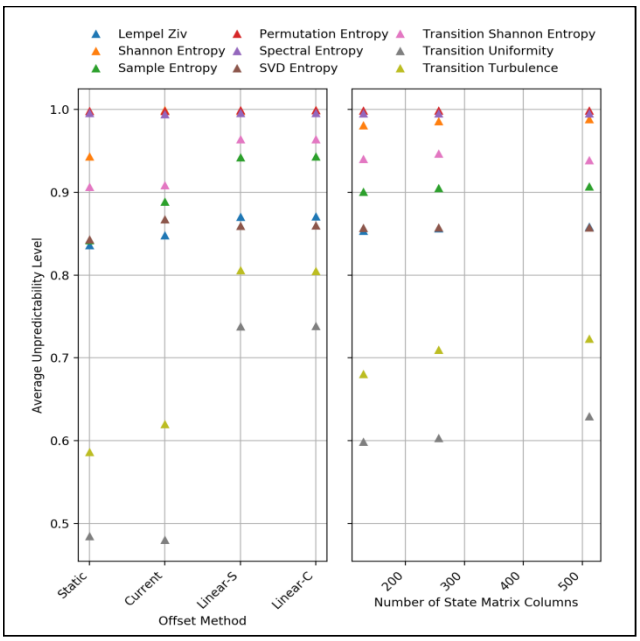


Fig. 6. Comparison of unpredictabilty metrics, averaged over all address sequences, when varying the offset method (*left*) and number of state matrix columns (*right*). The offset methods are arranged from lowest to highest overall unpredictability scores per TABLE IV.

A. *Fibonacci Experiment*

For the first experiment, we consider a hardware setup that is designed to calculate Fibonacci numbers[2]. Starting with the first two Fibonacci numbers (0 and 1), two nodes (i.e., BC and RT) calculate up to the 24th Fibonacci number (46,368). We chose this number as it is the largest 16-bit unsigned value which fits into a single DW. After reaching this value (termed a generation), the process resets with the first two Fibonacci numbers.

One node serves as the host, and the other node serves as the participant node. The host node sends two Fibonacci numbers (e.g., 8, 13), and the participant node sends the result back. The host uses the result and sends the next two Fibonacci numbers (e.g., 13, 21). The sequence continues until the host receives the 24th Fibonacci number.

1) *Hardware*

The experimental hardware setup consists of a single ENET2-1553 device from AltaData Technologies[3]. The device has 2 dual-redundant (i.e., two buses) channels. We used one channel for the BC and RT and another for the BM as shown in Fig. 7. We connected the two channels to the same bus using a bus coupler (also from AltaData) on bus A. Bus B was not used for our experimentation.

[2] Calculation of Fibonacci numbers is selected merely as an illustrative example and to avoid discussion of real but potentially sensitive processes.

[3] https://www.altadt.com/product/enet2-1553/

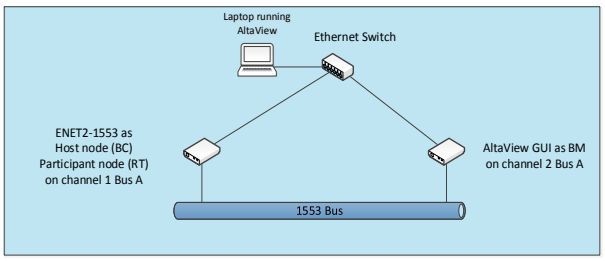


Fig. 7. Fibonacci experimentation setup

2) Software

We used the AltaAPI dynamic link library (dll) with our C++ codebase. The application programming interface (API) allowed the computer to read and write to the bus as needed. We used a multi-threaded approach to mimic different devices on the bus. In other words, each node (host, participant) ran in a separate thread. The entire Fibonacci calculation occurs via communication between these devices over the MIL-STD-1553 bus. Each node (i.e., host, participant) uses the dll to read and write to the bus. Effectively, the host node serves as a BC and the participant node serves as a RT. We use the tool AltaView (a graphical user interface (GUI) tool) to capture all traffic on the bus by serving as a BM.

3) Performance Evaluation

We evaluated the code size, size of the state matrix, and time required to generate a state matrix to evaluate the feasibility of implementing MTD in a real-time environment. We further evaluated the reliability of the algorithm (i.e., calculation of the correct Fibonacci sequence), and additional messages created and sent when MTD is used. The last quantity is used to determine if the inclusion of MTD could cause traffic congestion and unacceptable delays (i.e., overhead).

B. Cyber Resilience Experiment

For the second experiment, we consider a scenario in which an attacker is performing reconnaissance on the MIL-STD-1553 network and is attempting to exfiltrate data. We assume a similar network and the Fibonacci process as described in the previous experiment. We further assume an attacker is on the bus listening to all traffic to and from a specific address that the attacker believes corresponds to a "high-value", target node.

Without MTD the adversary can listen and capture all traffic to and from the target node. With MTD, the adversary is expected to capture less information since the adversary is not expected to know how the address changes. We quantify how much less information the adversary captures when using MTD.

1) Adversarial Model

In our experiment, we assume the adversary has direct access to the bus and can read/monitor all data on the bus. The adversary does not know the secret key shared offline. The adversary does not know the address of the target node, so it guesses as to the starting address.

We assume two different types of adversaries. The "static" adversary continues to listen at the same address for the entirety of the experiment. The "learning" adversary can detect when an MTD update occurs and then requires a certain number of frames to identify the target node's new address.

2) Hardware

We utilize the same hardware as the previous experiment, but we add another ENET2-1553. In this arrangement, we have 4 channels of MIL-STD-1553. We utilize two channels. One channel has a host node (BC), participant node (RT), and listening node (BM). The second channel has what we call the exfiltration node (exfil BM) as shown in Fig. 8.

3) Software

The software uses the same AltaAPI dll to connect with the ENET2-1553 devices. For this experiment, we do not use the AltaView tool as we programmatically capture data for both the regular BM and exfil BM.

4) Cyber Resilience Metrics

We measure the number of messages from the BC to the target node that the exfiltration listening BM contains at the end of the experiment. This quantity is measured with and without the MTD algorithm running. We quantify the cyber resilience benefit of the MTD by comparing the quantities resulting when MTD is used and not used.

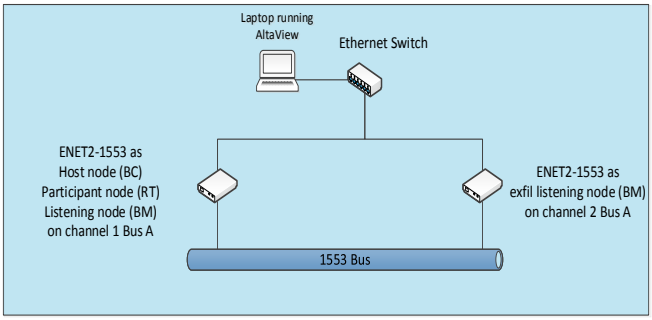


Fig. 8. Adversarial cyber resilience experiment setup

5) Experimental Design

To capture the stochastic variability, we use the following experimental design. For the static adversary,

1) The target node starts at address 1.
2) Set the MTD update frequency to once per frame.
3) Randomly select an address (0-31) for the exfiltration BM to listen to. Each address is assigned equal probability of being chosen (1/31).
   a) Perform 1000 generations of the Fibonacci calculation.
   b) Compare the number of messages found on the exfiltration BM to the number of actual messages to the target node found on listening BM.
4) Repeat steps 3, 3a, and 3b 24 more times, for a total of 25 trials.
5) Repeat steps 2) to 4) but change the update frequency to once per N frames where N = 6, 11, 16,…, 96.

For the learning adversary, we use the following process:

1) The target node starts at address 1.
2) Set the MTD update frequency to once per frame.
   a) Perform 1000 generations of the Fibonacci calculation.

b) Assume the adversary requires L=2 frames to learn the target nodes new address.
c) Calculate the ratio of total messages sent to the target vs. the number of messages that the adversary missed because it took L frames to learn the new address.

3) Repeat steps 2b and 2c for L = 4,8, 16, 32, 64, 128
4) Repeat all steps 2-3 but change the update frequency to once per N frames where N = 6, 11, 16,…, 96.

VI. Analysis and Results

A. MTD Algorithm Performance

The first experiment allowed us to determine various attributes about the codebase of the MTD algorithm. The second algorithm gave us the ability to quantify the benefit of using MTD.

1) Code Size

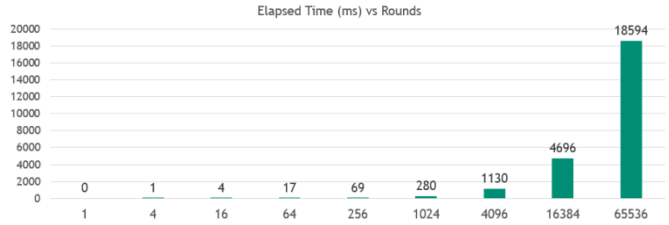
We compiled the MTD algorithm in C++ to a dll in debug mode. The unoptimized code size is 1.6 MB. Many general-purpose computing systems will have plenty of space to hold the dll. For smaller processing systems (i.e., microcontrollers), the code size may be a concern. Another aspect of code size is the size of the state matrix. The size depends on the number of columns generated. While the array is only 31 numbers, we calculated it based on 32 bytes for a single column due to the fact most memory would be allocated using a multiple of 2. For our work, we limited the number of columns to 65,536. At this size the state matrix occupies 2 MB as show in TABLE V.

TABLE V. Size of State Matrix

Number of Columns	Approximate Size (KB)
1	0.03125
4	0.125
16	0.5
64	2
256	8
1024	32
4096	128
16384	512
65536	2048

2) State Generation

In addition to the size of the code and state matrix, the performance needed to generate the state is another consideration. We benchmarked state generation using a modern desktop computer[4] desktop computer to determine the speed of the algorithm. We determined that the maximum amount of time needed for state generation is slightly under 19 seconds as shown in Fig. 9. Both measuring directly and computing directly (18594 ms / 65536 rounds) showed the speed to generate a column requires at least 283 microseconds as shown in Fig. 10. Slower processors will require more time, so this suggests that the state matrix should be generated before usage.



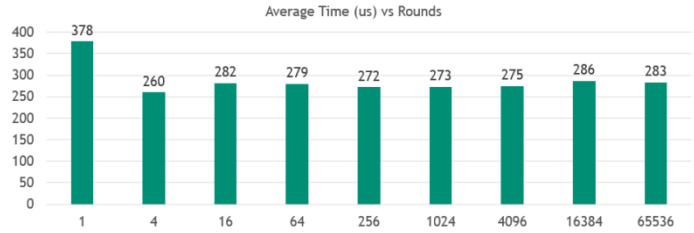


Fig. 9. Time to generate the state matrix

Fig. 10. Time to generate a column of the state matrix

3) Reliability

The two nodes correctly calculated the 24th Fibonacci number in every experiment that we conducted. Even when resetting the numbers back to 0 and 1 (i.e., start a new generation), the nodes calculated the correct number (0xB520 or 46,368). This accuracy gives us confidence in the reliability of the MTD algorithm. As mentioned before MTD does come with a cost. The MTD commands do consume time on the bus that would otherwise be idle. For this simple experiment, a frame consisted of 2 messages:

- Message #1: Fibonacci numbers sent from host to participant node
- Message #2: Next Fibonacci number in the sequence sent from the participant node to the host node

When we add the MTD update command, we add one message to the two-message frame. This incurs an overhead of 50%. If we send an MTD update every 2 frames (4 messages), we have an overhead of 25%. Therefore, the system designer has the option to send an MTD update every X frames as shown in TABLE VI. We use the following formulate to calculate overhead:

$$\frac{(\#\text{ of messages with MTD}) - (\#\text{ of messages without MTD})}{\#\text{ of messages without MTD}}$$

We multiply the result by 100 to get the percent overhead. As the reader can see, the impact of overhead to the system can be reduced or increase based on the frequency of sending the MTD update command.

[4] Intel Core i7-6700 CPU @ 3.40 GHz

TABLE VI. MTD FRAME OVERHEAD

MTD Frequency (# of frames between Update commands)	Framing Overhead (%)
1	50.0
2	25.0
3	16.7
5	10.0
10	5.0
20	2.5
50	1.0
100	0.5

B. *Cyber Resilience Evaluation*

Fig. 11 displays the results for the static adversary. Across all frequencies, the average fraction of messages exfiltrated is about 0.032, which is approximately 1/31, the pink horizontal line in Fig. 11. This result matches the theoretical expectation. If the adversary randomly selects an address to listen to and stays at that address, the target node will be at that address for approximately 1/31 of the messages.

This result is somewhat counterintuitive because one would expect more frequent MTD updates to increase resilience. However, because we assume the static defender listens at the same address for the entire time, we cannot observe the impact the frequency of MTD updates has on a learning attacker.

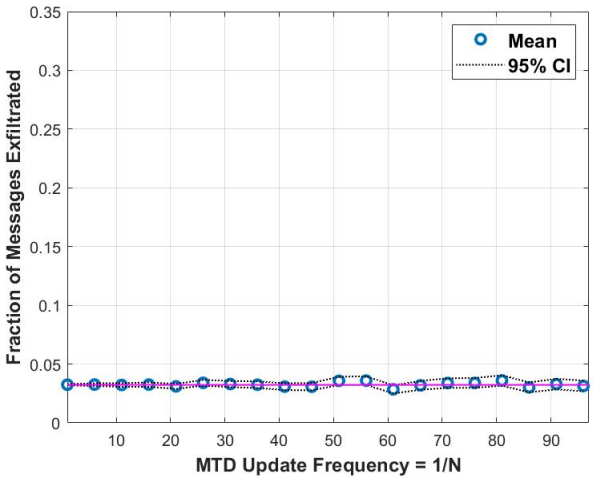


Fig. 11. Static Adversary Results: update frequency = once per 1, 6, …, 96 frames.

Fig. 12 displays the results for the learning adversary, and the different adversary learning rates are shown with different colors. In this figure, we can see the cyber resilience benefits of updating more frequently. When the update frequency is faster than the learning frequency, no messages are exfiltrated. When the update frequency is slower than the learning rate, we observe that a smaller difference between update frequency and learning rates results in fewer exfiltrated messages.

For example, when the MTD update occurs every 10 frames and the adversary has a learning rate of 8 (i.e., 8 frames pass before the adversary learns the new address), the adversary exfiltrates 2 out of 10 messages on average. If the adversary can double its learning rate to 4 (i.e., 4 frames pass before the adversary learns the new address), the adversary can exfiltrate 6 out of 10 messages on average. (Note that for several MTD update frequency- learning rate combinations, no messages are exfiltrated. In these instances, the yellow dot (corresponding to learning rate of 128 frames) may be on top of the other colored dots.)

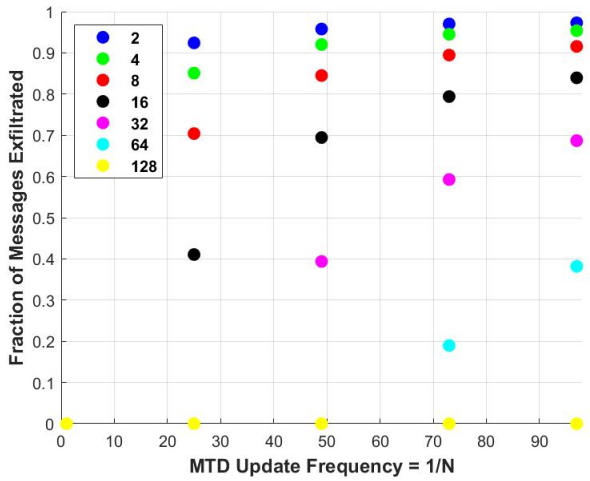


Fig. 12. Learning Adversary Results: MTD update frequency = once per 1, 26, 51, 76, 101frames. Adversary learning rates denoted with different colors. L= implies that X frames must be sent for the adevsray to learn the new address.

From these results, we conclude that MTD can significantly (~97%) reduce the amount of messages exfiltrated by the static adversary. Against a learning adversary, more frequent MTD updates are more effective at decreasing exfiltration. If the update occurs more frequently than the adversary's learning rate, exfiltration may be prevented entirely.

VII. CONCLUSIONS AND FUTURE WORK

The MTD algorithm worked well for MIL-STD-1553 and its bus-based topology. Other protocols such as CAN bus should work behave similarly as they are bus-based protocols as well. Non-bus protocols such as Spacewire (switched-based topology) would provide a different set of challenges to integrate the algorithm [19]. Synchronization in switched-based topologies may require a reliable transport at a higher protocol level.

While our focus has been on real-time protocols, the MTD algorithm has no assumption of the environment. It could be applied to non-real-time systems. For example, one could interpret the various columns as keys. A system could use the index value to synchronize which keys all participating nodes use at a given moment in time.

Our experimentation requires a host node. For MIL-STD-1553, the BC is a natural host node. For CAN bus, all nodes can

send at the same time and arbitration is built into the protocol. How is a host node selected for these protocols? Is a host node needed? Furthermore, peer-to-peer networks may need to operate without a host node. These concerns remain an open area of research.

Last, we focused mainly on the usage of the algorithm. There are other research topics we could pursue to improve the performance of the algorithm as well as its entropy. For example, how many columns are required to encounter every address once? Maybe there is not a need for 65,536 columns. Is there an optimal number of columns? Does this optimal value change if we use a different offset mechanism? In general, there are more research questions we can pursue specific to optimizing the algorithm and its design.

ACKNOWLEDGEMENT

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. This work is supported by the Laboratory Directed Research and Development program of Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DENA0003525.

REFERENCES

[1] M. Hegarty. "MIL-STD-1553 Goes Commercial," Data Device Corporation. June 2010.

[2] B. W. Gensch. "Evolving Moving Target Defense Configurations," UMM CSci Senior Seminar Conference, May 2016 Morris, MN

[3] N. Ahmed, B. Bhargava: Mayflies: A Moving Target Defense Framework for Distributed Systems. Proceedings of ACM Conference on Computers and Communications Security (CCS) workshop on Moving Target Defense: 59-64, Vienna, October, 2016

[4] N. Ahmed and B. Bhargava. "Bio-inspired Formal Model for Space/Time Virtual Machine Randomization and Diversification," IEEE Transactions on Cloud Computing, April, 2019

[5] N. Ahmed, B. Bhargava. "Towards Targeted Anomaly Detection Deployments in Cloud Computing," IEEE International Conference on Cloud Computing and Services Science. Lisbon, Portugal, May 2015.

[6] Chavez, W. Stout, and S. Peisert. "Techniques for the Dynamic Randomization of Network Attributes," Proceedings of the 49th Annual IEEE International Carnahan Conference on Security Technology, 2015.

[7] K. A. Farris and G. Cybenko, "Quantification of moving target cyber defenses," in Proceedings Volume 9456, Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security, Defense, and Law Enforcement XIV, 2017, p. 94560L94560L.

[8] S. Jones, A. Outkin, J. Gearhart, et al. "Evaluating Moving Target Defense with PLADD," SAND2015-8432R, Sandia National Laboratories, Albuquerque, New Mexico.

[9] S. Hossain-McKenzie, C. Lai, A. Chavez, E. Vugrin, "Performance-Based Cyber Resilience Metrics: An Applied Demonstration Toward Moving Target Defense", IECON 2018 Proceedings.

[10] United States Department of Defense. "Digital Time Division Command/Response Multiplex Data Bus", February 2018.

[11] Stan, Orly, et al. "Intrusion detection system for the mil-std-1553 communication bus." IEEE Transactions on Aerospace and Electronic Systems 56.4 (2019): 3010-3027.

[12] Woo, Samuel, et al. "Can id shuffling technique (cist): Moving target defense strategy for protecting in-vehicle can," IEEE Access 7 (2019): 15521-15536.

[13] Brown, Richard, et al. "Dynamic Address Validation Array (DAVA) A Moving Target Defense Protocol for CAN bus," Proceedings of the 7th ACM Workshop on Moving Target Defense. 2020.

[14] R. Vallat. 2020. EntroPy, ver. 0.1.2 [Online]. Available: https://raphaelvallat.com/entropy.

[15] A. Lempel and J. Ziv, "On the Complexity of Finite Sequences", *IEEE Transactions on Information Theory*, vol. 22, no.1, pp 75–81, 1976.

[16] J.S. Richman and J. R. Moorman, "Physiological time-series analysis using approximate entropy and sample entropy," *American Journal of Physiology-Heart and Circulatory Physiology*, vol. 278, no. 6, pp. 2039-2049, 2000.

[17] C. Bandt and B. Pompe, "Permutation entropy — a natural complexity measure for time series," *Phys Rev Lett*, vol. 88, no. 17, Apr 2002.

[18] A. Bramson, A. Baland, and A. Iriki, "Measuring dynamical uncertainty with revealed dynamics markov models," *Frontiers in Applied Mathematics and Statistics*,vol. 5, no. 7, Feb 2019.

[19] Star-dundee.com "SpaceWire's User Guide," Retrieved 27 October 2019