# Overshadow PLC to Detect Remote Control-Logic Injection Attacks

Hyunguk Yoo[1], Sushma Kalle[1], Jared Smith[2], and Irfan Ahmed[1,3]

[1] University of New Orleans, New Orleans LA 70148
{hyoo1,skalle1}@uno.edu
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37830
smithjm@ornl.gov
[3] Virginia Commonwealth University, Richmond VA 23221
iahmed3@vcu.edu

**Abstract.** Programmable logic controllers (PLCs) in industrial control systems (ICS) are vulnerable to remote control logic injection attacks. Attackers target the control logic of a PLC to manipulate the behavior of a physical process such as nuclear plants, power grids, and gas pipelines. Control logic attacks have been studied extensively in the literature, including hiding the transfer of a control logic over the network from both packet header-based signatures, and deep packet inspection. For instance, these attacks transfer a control logic code as data, into small fragments (one-byte per packet), that are further padded with noise data. To detect control logic in ICS network traffic, this paper presents `Shade`, a novel shadow memory technique that observes the network traffic to maintain a local copy of the current state of a PLC memory. To analyze the memory contents, `Shade` employs a classification algorithm with 42 unique features categorized into five types at different semantic levels of a control logic code, such as number of rungs, number of consecutive decompiled instructions, and n-grams. We then evaluate `Shade` against control logic injection attacks on two PLCs, Modicon M221 and MicroLogix 1400 from two ICS vendors, Schneider electric and Allen-Bradley, respectively. The evaluation results show that `Shade` can detect an attack instance (i.e., identifying at least one attack packet during the transfer of a malicious control logic) accurately without any false alarms.

**Keywords:** Control Logic · PLC · SCADA · Industrial Control System.

## 1 Introduction

Industrial Control Systems (ICS) actively control and monitor physical processes in critical infrastructure industries such as wastewater treatment plants, gas pipelines, and electrical power grids. Since the discovery of Stuxnet in 2010, an unseen nation-state malware that sabotaged Iran's nuclear facilities, the number of ICS vulnerabilities reported each year has been dramatically increased [18], and sophisticated attacks targeting critical infrastructure continue to occur [13, 14, 17, 20]. Designed to be isolated from the outside world, the

security of ICS environments was not a priority. However, these devices are increasingly becoming connected to corporate networks and the broader Internet for economic gain, more fluid business processes, and compatibility with traditional digital IT infrastructure [10, 22]. Unfortunately, their connectivity exposes vulnerabilities and unrestricted access by remote and insider attacks.

Within the ICS domain, Programmable Logic Controllers (PLCs) directly control a physical process located at a field site. Occurring in many ICS environments [19], PLCs are controlled by remote control center systems such as a human-machine interface (HMI) and engineering workstations via ICS-specific network protocols, such as Modbus, and DNP3 [7]. A PLC is equipped with a control logic that defines how the PLC should control a physical process.

Acting via channels exposed by the increasingly large threat surface of ICS networks and devices, attackers can target the control logic of a PLC to manipulate the behavior of a physical process. For instance, Stuxnet infects the control logic of a Siemens S7-300 PLC to modify the motor speed of centrifuges periodically from 1,410 Hz to 2 Hz to 1,064 Hz repeatedly, resulting in device failure. In most cases, the control logic of a PLC can be updated through the network using modern, yet typically not encrypted PLC communication protocols. Exploiting this feature, various classes of remote control logic injection attacks have been studied in the past, such as Stuxnet [8], Denial of Engineering Operations (DEO) attacks [24], and Fragmentation and Noise Padding [27]. Upon detection, a typical response may include blocking any transfer of control logic over the targeted network. For instance, Stuxnet compromises the STEP 7 engineering software [8] in a control center to communicate with a target PLC in a field site. Next, the malware transfers a malicious control logic program to the PLC. Notably, this attack can be prevented if ICS operators prevent the transfer of control logic over the network.

Recently, Yoo and Ahmed [27] presented two stealthy control logic injection attacks, referred to as *Data Execution*, and *Fragmentation and Noise Padding* to demonstrate that an attacker can subvert both packet-header signatures and payload inspection to transfer control logic to a PLC successfully. In the *Data Execution* attack, an attacker deceives packet header inspection by transferring control logic to data blocks of a target PLC and then, modifies the PLC's system control flow to execute the logic located in data blocks. Packet-header signatures do not prevent the data blocks because they contain sensor measurement values and actuator state, which are normally sent to the human-machine interface at the control center. In the *Fragmentation and Noise Padding* attack, an attacker sends a control logic to a PLC in small fragments (typically one byte per packet) and further adds a large padding of noise to evade traditional deep packet inspection.

These attacks give adversaries a significant advantage over operators relying on existing modern protections against network-based attacks, which utilize stealth-mechanisms. To that end, this paper presents a *first-of-its-kind system*, `Shade` to detect control logic in an ICS network traffic when an attacker employs both stealthy *Data Execution*, and *Fragmentation and Noise Padding* attacks to

compromise critical infrastructure networks. Shade observes ICS network traffic to maintain a local (shadow) copy of a PLC's memory layout using read and write messages from and to the PLC. Our system scans the shadow memory as an ensemble of supervised learning algorithms with 42 custom, domain-relevant features of control logic code categorized into five types. These types lie at different levels of semantics extracted from the PLC control logic code: 1) decompilation, 2) rung, 3) opcode identification, 4) n-gram, and 5) entropy.

We implement the attacks on two different vendors' PLCs, Allen Bradley's MicroLogix 1400 and Schneider Electric's Modicon M221. Note that these PLCs are originally utilized by Yoo and Ahmed to demonstrate the attacks [27], and we use them to recreate the attacks to evaluate the accuracy of Shade. Our evaluation results show that while the traditional payload inspection fails to detect these attacks, Shade can detect the transfer of all control logic programs accurately without any false alarms. Furthermore, Shade's performance overhead lies at 2%, a necessary trait for ease of deployment into real-world ICS networks.

**Contributions.** Our contributions can be summarized as follows:

- We validate two recent stealthy control-logic injection attacks that can subvert both protocol header signatures and deep packet inspection.
- We present Shade, which is a novel shadow memory approach to detect control logic code in ICS network traffic when the stealthy control logic attacks are employed.
- We study different types of the features on control logic code at different semantic levels of a control logic and identify a best set of feature to achieve optimal results, i.e., accurate detection of the transfer of control logic instances in an ICS network traffic without any false positives.
- We evaluate Shade on real-world PLCs used in industrial settings.
- We release our datasets and the source code of Shade[1].

**Roadmap.** We have organized the rest of the paper as follows: Section 2 provides the background. Section 3 presents the shadow memory-based control logic detection technique, followed by its implementation and evaluation results in Section 4 and Section 6 respectively. Section 7 covers the related work, followed by the conclusion in Section 8.

## 2 Background: Control Logic Injection Attacks

*Control logic* is a program which is executed repeatedly in a PLC. It is programmed and compiled using *engineering software* provided by PLC vendors. There are five PLC programming languages defined by IEC 61131-3 [11]: ladder logic, instruction list, functional block diagram, structured text, and sequential flow chart. A PLC is usually equipped with communication interfaces such as RS-232 serial ports, Ethernet, and USB to communicate with the engineering software so that control logic can be downloaded to or uploaded from a PLC.

In general, a control logic can be divided into four different blocks when transferred to or from a PLC: the configuration block, code block, data block, and information block. The configuration block contains information on the other

[1] https://gitlab.com/hyunguk/plcdpi/

blocks (e.g., the address and size of the blocks) and other configuration settings for the PLC (e.g., IP address of the PLC). The *compiled* code-block controls logic code running in the PLC. The data block maintains the variables (e.g, `input`, `output`, `timer`, etc.) used in the code block. Finally, engineering software uses the information block to recover the original project file from the decompiled source code when the control logic is uploaded to the engineering software.

In a typical control logic injection attack [8, 24], an attacker downloads malicious control logic onto a target PLC by interfering with the normal PLC engineering operation of downloading/uploading control logic. Stuxnet [8], a representative example of this type of attack, infects Siemens SIMATIC STEP 7 (engineering software) and downloads malicious control logic to target PLCs (Siemens S7-300) by utilizing the infected engineering software. The lack of authentication measures in the PLC communication protocols results in successful exploitation. In our experience, control logic downloading/uploading operations do not support authentication or authentication is only supported in one direction, either download or upload.

Recently, Yoo and Ahmed [27] presented two stealthy control logic injection attacks, referred to as *Data Execution*, and *Fragmentation and Noise Padding* to hide the transfer of control logic over the network from packet-header signatures and deep packet inspection. We now cover these attacks in detail.

**Data Execution Attack.** The Data Execution Attack evades network intrusion detection systems (NIDS) that rely on signatures based on packet header fields by transferring the compiled code of control logic to the data blocks of a PLC. The data blocks exchange sensor measurement values and states of PLC variables (e.g., `inputs`, `coil`, `timers`, and `counters`). Since control center applications (e.g., HMI) may frequently read and write on those data, the NIDS signatures must not raise an alarm for data blocks in the network traffic of ICS environments. Therefore, the attack evades the NIDS signatures by embedding attacker's logic code in data blocks. After transferring the logic code to a PLC, the attack further modifies the pointer to the code block to execute the attacker's logic located in data blocks. This code could contain instructions similar to Stuxnet, which would result in major ICS failures and costly repercussions. Most PLCs in the market do not enforce data execution prevention (DEP), thereby allowing the logic in data blocks to execute. However, this attack can be subverted by payload-based anomaly detection.

**Fragmentation and Noise Padding Attack.** This attack subverts payload-based anomaly detection by appending a sequence of padding bytes (noise) in control logic packets while keeping the size of the attacker's logic code in packet payloads significantly small. The ICS protocol often have address or offset fields in their headers, which are utilized by the attack to make the PLC discard the noise padding.

In their study [27], they showed that both signature-based header inspection and payload-based anomaly detection can be bypassed by an attack combining the two stealthy attacks, i.e., transferring attacker's logic code in a data block while fragmenting the code and appending a noise padding.

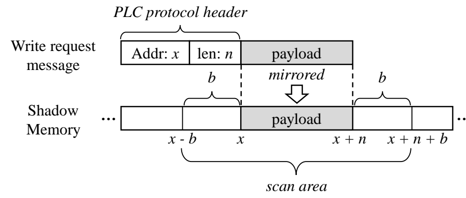



**Fig. 1.** Shadow memory scanning

## 3 Shadow Memory-based Control Logic Detection

### 3.1 Shade - a Shadow Memory Approach

Generally, shadow memory refers to a technique to store information on memory (e.g., whether each byte of memory is safe to access), which is utilized in most memory debuggers [6, 9]. In this paper, however, we present shadow memory as a mirrored space of the *protocol address space* of a PLC. We define the protocol address space of a PLC as the range of space that can be addressed for payload data through a PLC protocol. For example, if the write request message format of a PLC protocol has a fixed 2-byte address field that specifies the byte offset of data to be written, the address space of the PLC protocol will be 64KB.

The proposed approach referred to as `Shade` maintains shadow memory of each PLC and detects control logic code by scanning the shadow memory rather than the individual packet payloads. Briefly speaking, `Shade` works as follows: when a write request packet to a PLC is identified in an ICS network traffic, its payload data is reflected in the shadow memory. `Shade` uses packet-header values to map the data at a correct memory location of the shadow memory and excludes any excess data (such as noise) that resides in a packet payload but is not written to the PLC memory. Note that attacker can exploit protocol specifications to include noise data in a packet payload but does not write the noise to PLC memory to avoid any risk of crashing the PLC. In *Fragmentation and Noise Padding*, attacker manipulates header values to filter noise data when the packet arrives at the PLC. After mapping a payload to shadow memory, `Shade` scans the shadow memory to determine whether or not the control logic code resides in the memory. Even though each attack packet of the *Fragmentation and Noise Padding* attack contains a tiny size of code fragment with large noise, it will eventually composes a detectable size of code chunk in the shadow memory, thus making the proposed detection method effective.

Figure 1 depicts the mirroring and scanning of shadow memory. When a write request packet is identified, its payload is mirrored to shadow memory according to the packet's address and length fields. Then, we scan the surrounding space including the area where the payload is mirrored. We call the area scanned for each write request packet the *scan area*. The range of the scan area is determined by the payload size, the address of a write request packet, and the scan boundary parameter $b$. With the address $x$ and the payload length $n$, the lower bound of the scan area is defined by $MAX(0, x-b)$ and the upper bound is $MIN(m, x+n+b)$, where $m$ is the highest address of shadow memory.

Instead of scanning the whole memory, we propose to scan a small chunk of relevant shadow memory which is updated recently. This approach has two advantages: 1) Avoid overhead, and 2) Reduce false alarms.

**Avoiding overhead.** Scan of a small memory chunk avoids significant performance overhead. As we will show in Section 6, the overhead of shadow memory scanning is 31.92% for the Schneider Electric's M221 PLC with a boundary parameter of 236 and 701 bytes of scan range in average. If we perform a full-scanning for the M221 PLC, of which the shadow memory size is around 64KB, the overhead will be unfeasible for a real-world deployment.

**Reduce false alarms.** Worse, full-scanning more often produces false positives. If a non-code packet (e.g., a packet containing PLC variable data or configuration information) is misclassified as a code packet, all the following non-code packets will be misclassified as well unless the mirrored payload data of the initially misclassified packet is removed from the shadow memory. However, clearing a certain area of shadow memory makes the shadow memory inconsistent with the actual state of a PLC, which could lead to failure of detecting attack packets containing fragmented code later.

With partial-scanning, the scan boundary parameter $b$ is a trade-off factor. Increasing $b$ would raise the true positive rate (increase sensitivity) but also raise the false positive rate and performance overhead, and vice-versa. Assume that $n$ is the minimum size of the code fragment that can be detected by a classification algorithm $C$, and $k$ is the maximum payload size, then $n$ must be smaller than $k$ if the classification algorithm $C$ has high sensitivity for detecting logic code in a *packet payload*.

With shadow memory, if a code chunk in the shadow memory is larger than or equal to $n$, setting $b$ with $k$ ensures that the classification algorithm $C$ can detect it. Let's assume that $(n-1)$ bytes of code in shadow memory from address $x$ to $(x+n-1)$, and one-byte of attacker's code fragment is being written to shadow memory. If the one-byte code is written at the address $(x-1)$ or $(x+n)$, then the size of code chunk will be $n$ bytes which can be detected. Note that the attacker can write different parts of a code in a random sequence, which may delay the detection. However, the consecutive code size will end up exceeding $n$-bytes of code chunk in the scan area and is detected by `Shade`.

### 3.2 Feature Extraction with Different Semantic Levels

In the training phase, `Shade` extracts 42 different features 1 from the scan area of shadow memory, then it selects only the best features for training a classifier. Figure 2 highlights the different features with varying semantic levels studied in this paper. N-gram or entropy does not require any syntax or semantic knowledge of the underlying data. On the other hand, features such as the number of the identified opcodes, the number of the rungs, and the number of successfully decompiled bytes require knowledge about the format and semantics of the data.

**Decompilation of Control Logic Code.** The feature *#dec* represents the longest length of a byte sequence which is successfully decompiled. Decompilation starts from each byte position in the scan area, recording the length of a

**Table 1.** Extracted features

| Feature | Description |
|---|---|
| *#dec* | The maximum length of decompiled byte sequence |
| *#op* | The number of the identified opcodes |
| *#rung* | The number of the identified rungs |
| *#Ngram* | The number of the n-grams that are present in a bloom filter ($1 < n \leq 20$) |
| *LNgram* | The longest *continuous* match of n-grams that are present in a bloom filter ($1 < n \leq 20$) |
| *entropy* | The byte entropy of scan area |

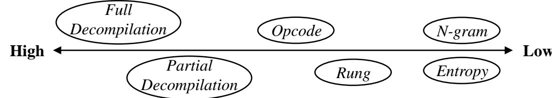



**Fig. 2.** Varying level of semantic knowledge on control logic code

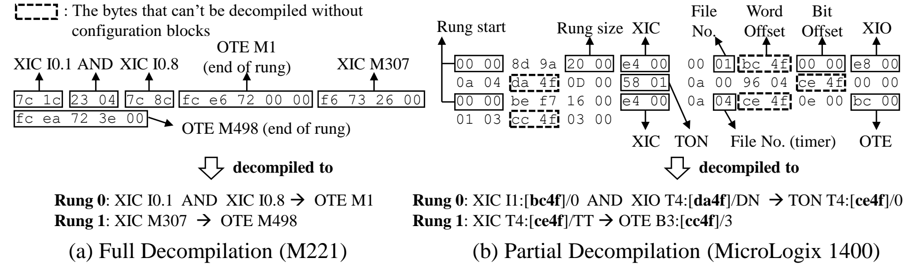
decompiled byte sequence for each position. Then, the longest length is selected for the *#dec* feature.

Several studies [1, 3] utilize a disassembler to detect x86 machine code in network traffic. Our decompilation approach is in some ways similar to those studies. Decompilation of control logic code has a unique characteristic compared to decompilation of a binary used in common IT systems. When compiling high-level language code (e.g., C/C++, Java) to low-level code (e.g., machine code, bytecode), finding the original structure of the high-level language code from the low-level code is non-trivial due to compiler optimizations. On the other hand, compilation of control logic code is performed in a manner that it is completely reversible, i.e., decompilation of logic code recovers the exact source code. This interesting design feature of control logic compilers makes it possible for the engineering software to show the original source code to PLC programmers or ICS operators when the control logic is retrieved from a PLC.

In our experience (with two engineering software, RSLogix and So-machine basic of two different vendors, Allen-Bradley and Schneider Electric on two PLCs, MicroLogix 1400 and M221), we find substantial one-to-one mappings between the high-level language code of two PLC languages (i.e., Ladder Logic, and Instruction List) and their (compiled) low-level code. This key discovery allows `Shade` to utilize a substitution table for decompilation. The extent of decompilation can differ between the engineering software. In some cases, decompilation of control logic code not only requires code blocks but also configuration blocks. For example, operands of instructions in code blocks may only be offsets from base addresses, with the base addresses stored in configuration blocks. If configuration block is not available, the full-decompilation is impossible. In these cases, `Shade` performs partial-decompilations.

Figures 3(a) shows an example of a full-decompilation for the Modicon M221 PLC. Each high-level language code is always mapped to the same low-level representation. For example, a Ladder Logic instruction `XIC I0.1` (examine if `input 1` is closed) is always mapped to (or compiled to) its low-level representation `0x7c1c` which is an RX630 machine instruction [26].

**Partial Decompilation of Control Logic Code.** On the other hand, a full-decompilation of the MicroLogix 1400 logic code requires additional information from a configuration block as well as code blocks [24]. Since an at-




(a) Full Decompilation (M221) (b) Partial Decompilation (MicroLogix 1400)

**Fig. 3.** Examples of full and partial decompilation of control logic

tacker does not necessarily send configuration blocks to a PLC for control-logic injection, we should not assume that `Shade` can obtain the configuration blocks for decompilation. Figure 3(b) shows an example describing how `Shade` performs a partial-decompilation for the MicroLogix 1400 PLC. There are two rungs (`Rung 0` and `Rung 1`), each of which starts with the rung start signature `0x0000` in low-level code. In `Rung 1`, `Shade` decompiles `0xe400` to `XIC` since the opcodes of Ladder Logic instructions of MicroLogix 1400 are always mapped to the same low-level code [24]. The system also knows the operand type of the `XIC` instruction is the `timer` object based on its file number (`0x04`). However, since control logic can have multiple timer objects, `Shade` does not know the timer object that corresponds to the `XIC` instruction. The engineering software of the MicroLogix 1400 PLC calculates the exact operand by $(Word\ offset - Base\ address)/Size\ of\ object$ where the base addresses of each object type is stored in the configuration block.

`Shade` leaves the low-level codes of those operands but counts the number of decompiled bytes as if the corresponding bytes are decompiled when they are between decompiled byte sequences. In the above example, the 2-byte hex values highlighted in bold are the operands that can not be decompiled without the configuration block. But `Shade` counts the number of decompiled bytes as 54 (the total size of `Rung 0` and `Rung 1`), even though parts of operands are not actually decompiled. Importantly, the purpose of decompilation in `Shade` is not recovering the source code, rather counting the number of decompiled bytes.

**Opcode & Rung Identification.** Control logic code consists of one or more rungs and a rung consists of one or more instructions. Typically, a rung has input (e.g., Examine-if-closed:XIC, Examine if open:XIO) and output (e.g., Output energize:OTE, Output Latch:OTL, Timer-on-delay:TON) instructions where a logical expression of the inputs is evaluated and the state of the outputs are changed based on the evaluation result in each PLC scan cycle.

To count the number of opcodes, `Shade` finds all occurrences of opcodes in the scan area, utilizing a table containing the mapping of opcodes between high-level code and low-level code. Unlike decompilation, the opcode identification does not utilize other semantics of logic code (e.g., rung structures). The rungs of control logic code are identified based on the knowledge of rung structures. In the case of MicroLogix 1400, rungs explicitly start with a signature (`0x0000`), followed by a field specifying the size of rung. On the other hand, the logic code

of the Modicon M221 PLC can be separated into rungs in a different way. We will discuss this in detail later in Section 4.

**N-gram Bloom filter & Entropy.** A common method in natural language processing, N-gram analysis extracts features from data without any semantic knowledge or the format of data. This method has been employed in a variety of applications, including packet payload inspection [2, 4]. Two primary approaches allow the construction an n-gram feature space from a packet payload: 1) counting the frequency of each n-gram, 2) counting the n-gram membership in a pre-defined n-gram set

*Counting the n-gram frequency.* In this approach, payload data is embedded in a vector space of $256^{\text{n}}$ dimension where $n$ is the size of n-gram. This approach suffers when $n$ is greater than 1, resulting in a sparse matrix being used for training a classifier. This is due to the large vector space compared to the typical payload sizes of PLC protocols[2].

*Counting the n-gram membership in a pre-defined n-gram set.* The second approach counts the number of n-grams that present or absent in a pre-defined n-gram set. For example, Anagram [4] stores all the unique n-grams of the normal packet payload in a bloom filter in the training phase, then counts the number of the n-grams of a testing packet payload that are absent in the bloom filter, to score the abnormality of each packet. Fortunately, the feature space of data in this approach is simply one-dimensional regardless of the size of the n-gram, which allows a higher order n-gram to be used. Generally, a high order n-gram ($n > 1$) is more precise than a 1-gram to detect anomalous packets. This approach also provides more resistance against mimicry attacks [4].

Building off its advantages, `Shade` employs the latter approach to extract two different types of n-gram features (i.e., *#Ngram*, *LNgram*). Before training a classifier, `Shade` stores in bloom filters all the unique n-grams of *normal* write request message payload containing logic code, for each n-gram size ($1 < n \leq 20$). Then, `Shade` extracts two different types of n-gram features utilizing the bloom filters: 1) the number of n-grams *present* in the corresponding n-gram bloom filter, 2) the maximum number of *consecutive* n-grams in the bloom filter.

For the *entropy* feature, `Shade` calculates the Shannon Entropy of the byte value of the payload data.

### 3.3 Feature Selection & Classification

In the training phase, we evaluate each feature individually using a one-dimensional Gaussian Naive Bayes classifier [21] to select the best features for generating classification models. We employ two classic, explainable machine learning algorithms: 1) Gaussian Naive Bayes and 2) Support Vector Machine (SVM). With these algorithms, we then generate classification models and compare the detection performance of each. As we will show in Section 6, neither algorithm performs significantly better than the other. Critically, however, the use (or non-use) of shadow memory contributed significantly to the success of attack detection within the ICS environment.

[2] The maximum payload sizes are 236 bytes and 80 bytes for the Modicon M221 PLC and the MicroLogix 1400 PLC, respectively

# 4 Implementation

We implement `Shade` for two different vendors' PLCs, Schneider Electric Modicon M221 and Allen Bradley MicroLogix 1400. To demonstrate the effectiveness of `Shade` against both *Data Execution*, and *Fragmentation and Noise Padding* attacks, we evaluate and compare both `Shade` and traditional deep packet inspection (DPI). Specifically, `Shade` extracts features from the scan area of shadow memory while the DPI extracts them from the packet payload. To allow easy reproducibility, we leverage Python using the open-source Scapy packet manipulation library.

## 4.1 Shade implementation for the M221 PLC

**M221 Opcode & Rung Identification.** To identify opcodes in the M221 logic code, we developed a table which maps the opcodes of Instruction List to its low-level code. For the rung identification, we utilize the following rule of rung structure applied to the M221 logic code. We separate rungs in two cases: 1) after an output instruction (e.g., `ST %Q0.0` to energize `coil 0`) directly followed by an input instruction (e.g., `LD %I0.0` to examine if `input 0` is closed), 2) after the signature, `0x7f1a11`, which represents the end of block.

**Full-Decompilation of M221 Logic Code.** Since all the necessary information to recover the original source code is contained in the code block of the M221 control logic, `Shade` can perform a full-decompilation. Along with a substitution table which maps Instruction List code to its low-level code, `Shade` employs the knowledge of the code's block structure.

The M221 logic code contains three types of blocks: function blocks, comparison blocks, and operation blocks. The M221 PLC uses pre-defined function blocks such as `TON(Timer On-Delay)` and `CTU(Counter up)`. A function block starts with the signature `0x7f1a10`. The comparison blocks provide relational operations (e.g., $=$, $\leq$), while the operation blocks provide arithmetic and logical operations. They start with signatures `0x7f1aXX` where the third byte indicates the operator and operand type (e.g., addition with integers: `0x04`; addition with floating-point numbers: `0x39`).

Figure 4(a) shows an example of decompilation of a simple operation block. The first three bytes `0x7f1a3c` indicate an operation block performing division with two floating-point numbers. The sixth and seventh bytes (`0x32`) indicate two source operands as float *variables*. The following byte sequences `0x0281`, `0x0481`, and `0x0681` are decompiled to corresponding float type variables, `%MF1`, `%MF2`, and `%MF3` respectively. The recovered source code indicates that the result of `%MF2` divided by `%MF3` is assigned to `%MF1`.

When operation blocks are nested, a temporary variable encoded as `0xc290` is involved. This temporary variable is only visible in low-level code and we will refer to it as `TEMP`. Figure 4(b) shows an example of decompilation of nested operation blocks. The third byte in the first line of low-level code denotes the first operation block as multiplication with floating-point numbers. The seventh byte (`0x29`) indicates that the second operand on the right-hand side of the assignment operator is a float type *constant*. The next two bytes (`0xc290`), used for a destination operand, are converted to a `TEMP` variable. Then the last four

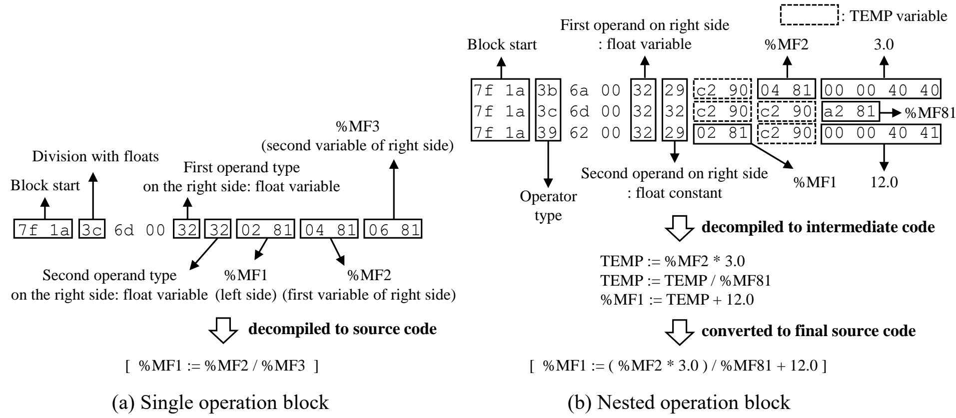



**Fig. 4.** Decompilation of operation blocks of M221 control logic code

bytes, `0x00004040` (little-endian), are converted to 3.0 by the IEEE 754 standard. In this manner, we first convert the low-level code to an intermediate code representation using the `TEMP` variable, which we then convert to the final source code variant.

**M221 Shadow Memory.** The proprietary protocol used in the M221 PLC has two 2-byte size address fields: the address type and the address fields [27]. `Shade` dynamically allocates 64KB of shadow memory space for each address type when the PLC first use an address type in a write request message. Given that the M221 PLC only employ a few designated address types, the size of the shadow memory always lies between 64KB to 320KB.

### 4.2 Shade implementation for the MicroLogix 1400 PLC

**MicroLogix 1400 Opcode & Rung Identification.** To identify opcodes in the MicroLogix 1400 logic code, `Shade` then utilizes a mapping table developed in [24]. With insight derived from this mapping, `Shade` maps the opcodes of ladder Logic to their low-level byte code. In a similar fashion, when we conduct rung identification, the rung starts with the signature (`0x0000`) and the rung size field are utilized. From the start address of the scan area, `Shade` automatically searches all rung start signatures. When `Shade` discovers a rung's start-signature, it checks if the size of the rung is at least 8 bytes, the minimum rung size of the MicroLogix 1400 logic code. Then, if the offset of the next rung's start-signature is correct (i.e., offset of the current signature + size of the current rung), `Shade` counts the current rung as a valid rung.

**Partial-Decompilation of MicroLogix 1400 Logic Code.** Recall that, since the full-decompilation of MicroLogix 1400 logic code requires configuration blocks as well as code blocks, `Shade` performs a partial-decompilation. `Shade` starts decompilation from each byte position in the scan area. Decompilation from a byte position ends in one of following conditions: 1) `END` instruction (which indicates the end of code), 2) undefined opcode, 3) invalid operand, and 4) invalid rung structure. The first and second conditions are straightforward. For the third condition, `Shade` can verify the bit offset of operands, although it cannot verify the

word offset (due to the lack of configuration blocks). Since the size of data type addressed by the word offset is 16-bit, the valid range of bit offset should be between `0x0000` to `0x000f`. On the other hand, the validity of a rung structure is checked by examining if the rung size is correct and the rung contains at least one instruction.

**MicroLogix 1400 Shadow Memory.** The PCCC protocol used in the MicroLogix 1400 PLC has four fields for addressing [24]: file number, file type, element number, and sub-element number. `Shade` allocates 64KB of shadow memory space for each {*file number, file type, element number*} tuple. Basically, each 64KB of shadow memory space corresponds to a specific file[3], since the element number is always `0x00` and the sub-element can be up to 2-byte size.

## 5 Description of Datasets

Table 2 describes the datasets that will be evaluated later in Section 6. We generate these two datasets for two different, but widely-deployed vendors' PLCs: Schneider Electric's Modicon M221 and Allen-Bradley's MicroLogix 1400 PLCs, using corresponding engineering software, SoMachine Basic v1.6 and RsLogix 500 v9.05.01 respectively. We contribute and evaluate four distinct datasets for each PLC i.e., training and attack datasets for both DPI and `Shade`. The datasets in this evaluation are modeled after the network packet datasets used in [27], which allow us to conduct a fair evaluation on `Shade`. The network packet datasets contain 51 and 127 unique control logic programs written in Ladder Logic and Instruction List for MicroLogix 1400 and Modicon M221 PLCs respectively[4] [27], among them 22 and 52 (binary) programs of each Modicon M221 and MicroLogix 1400 are used to generate bloom filters, while the rest are used to generate our novel datasets.

Based on the training datasets ($DS_{M221/ML1400,Packet/Shadow}$) which do not involve any evasion attacks, we use a *supervised learning* approach for our classification task to distinguish code and non-code packets. Note that our goal is not to distinguish malicious/benign logic but to identify all the control logic being transferred over the network even if evasion attacks are engaged. Accordingly, the control logic programs themselves in our datasets are not specially malicious. They are just numbers of unique control logic programs with varying complexity generated to encompass as many different characteristics of control logic programs as possible.

In our evaluation scenario (Section 6), we assume that any attempt to download control logic to a PLC is a *malicious action* that should be recorded or alarmed. This approach is particularly reasonable in the ICS domain because usually control logic update of a PLC is a rare event. Therefore, ICS operators

[3] In Allen-Bradley PLCs, each control logic block is called as a file

[4] The control logic programs were collected in two ways: 1) Generated in a lab environment using venders' engineering software and PLCs 2) Downloaded from various sources on the Internet (e.g., plctalk.net). Collectively, they are written for different physical processes (e.g., traffic light system, elevator, gas pipeline, hot water tank) with varying instructions and rung complexity

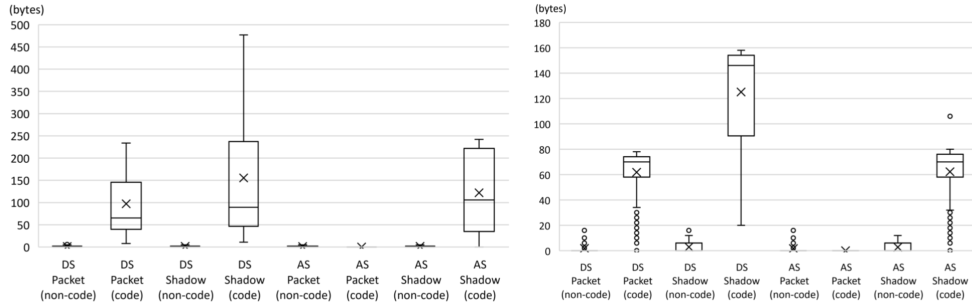



**Fig. 5.** Population of decompiled bytes (Left: Modicon M221, Right: MicroLogix 1400)

**Table 2.** Description of the datasets

| Datasets | # of write req. packets | # of packets of logic code | Avg. # of scanned bytes | Avg. # of dec. bytes (non-code) | Avg. # of dec. bytes (code) |
|---|---|---|---|---|---|
| $DS_{M221,Packet}$ | 1535 | 38 | 216 | 1.3 | 97.4 |
| $DS_{M221,Shadow}$ | 1535 | 38 | 679 | 1.5 | 155.2 |
| $AS_{M221,Packet}$ | 5362 | 3865 | 231 | 1.3 | 0.2 |
| $AS_{M221,Shadow}$ | 5362 | 3865 | 701 | 1.5 | 121.9 |
| $DS_{ML1400,Packet}$ | 5,465 | 684 | 52 | 1.9 | 61.8 |
| $DS_{ML1400,Shadow}$ | 5,465 | 684 | 170.7 | 2.9 | 125.1 |
| $AS_{ML1400,Packet}$ | 29,647 | 24,866 | 40.6 | 1.7 | 0 |
| $AS_{ML1400,Shadow}$ | 29,647 | 24,866 | 185.8 | 2.8 | 62.2 |

want to be informed of the existence of *any* control logic code in the network traffic for further decision making or forensic analysis. It would be also worth to mention that our approach complements control logic verification techniques [16] to distinguish malicious/benign logic. Note that identifying control logic must be done to verify it.

### 5.1 M221 Datasets

**Training Datasets.** We generate the $DS_{M221,Packet}$ dataset based on the network captures of control logic *downloading* to a PLC, which does not involve evasion attacks (i.e., our approach performs the control logic download operation using PLC-specific engineering software), while we extract features from an individual packet. Next, we produce $DS_{M221,Shadow}$ from the same network captures, except we extract each feature from the *scan area of shadow memory* (employed in `Shade`), and not the packet payload.

The boxplot on the left side of Figure 5 displays the population of decompiled bytes in the Modicon M221 datasets. As shown, clear differences exist between non-code and code packets in both packet-basis DPI and `Shade`, providing intuition that distinguishing between code and non-code packets would be possible by either method if no evasion attacks are present.

**Attack Datasets.** The $AS_{M221,Packet}$ and $AS_{M221,Shadow}$ datasets involve both the Data Execution, and Fragmentation and Noise Padding attacks, while the former dataset is generated based on packet payload and the latter is based on the shadow memory. Unlike the training datasets, which do not involve any evasion attacks, the populations of decompiled bytes are entirely different between packet-basis DPI and `Shade`, as shown in the left boxplot in Figure 5. It implies

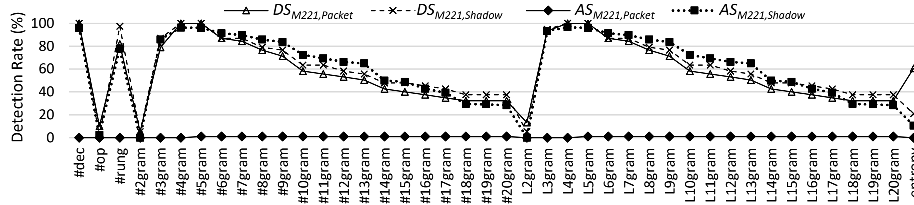



**Fig. 6.** The result of feature test on the Modicon M221 datasets

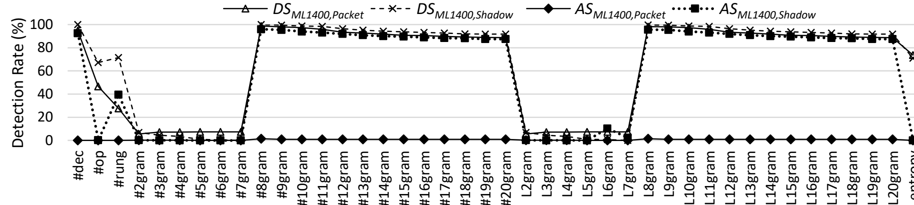



**Fig. 7.** The result of feature test on the MicroLogix 1400 datasets

that packet-basis DPI does not effectively identify code packets, but `Shade` does, based on the feature of decompiled bytes.

### 5.2 MicroLogix 1400 Datasets

**Training Datasets.** We generate the $DS_{ML1400,Packet}$ and $DS_{ML1400,Shadow}$ datasets based on the network captures of control logic downloading to a PLC, which again, does not involve any evasion attacks. Like the other PLC, one involves packet-basis DPI and the latter utilizes `Shade`.

**Attack Datasets.** The $AS_{ML1400,Packet}$ and $AS_{ML1400,Shadow}$ datasets then contain the Fragmentation and Noise Padding attacks, in the same manner as the prior PLC. We extract the features of $AS_{ML1400,Packet}$ from an individual packet payload (packet-basis DPI), while we extract those of $AS_{ML1400,Shadow}$ from the scan area of shadow memory (`Shade`).

Similar to the Modicon M221 attack datasets, we investigate the populations of decompiled bytes between code and non-code packets, finding a lack of distinguishable traits in the $AS_{ML1400,Packet}$ dataset. Converesely, their differences can be clearly identified in the $AS_{ML1400,Shadow}$ dataset. This difference can be shown in the right boxplot in Figure 5.

## 6 Experimental Evaluation

### 6.1 Feature Selection

In the training phase, we evaluate each feature in each dataset individually with a (one-dimensional) Gaussian Naive Bayes, selecting the most performant features for a binary-classification scenario. Figure 6 and 7 show the feature test results for Modicon M221 and MicroLogix 1400, respectively. The vertical axis represents the detection rate (true positive rate) at an unprecedented 1% false positive rate (FPR) for this particular classification problem.

Since no feature works effectively for the attack with packet-basis DPI datasets, we select features based on the rest of the datasets. The feature of the number

of decompiled bytes (*#def*) shows the highest performance among all the semantic features for both PLCs. On the other hand, *L4gram* is the best among the non-semantic features (i.e., n-grams, entropy) for Modicon M221 while *#8gram* is the best for MicroLogix 1400. Based on this result, the features of *#dec* and *L4gram* are selected for the Modicon M221 while *#dec* and *#8gram* are selected for the MicroLogix 1400.

Interestingly, specific size of n-gram features show similar or better detection rates than the *#dec* feature even though decompilation involves the highest semantic knowledge of control logic code, while n-gram does not require any semantic knowledge. Further analyzing the potential root cause, we found that some byte sequences in the non-code data were falsely decompiled, especially in the MicroLogix 1400 datasets. In the MicroLogix 1400 logic code, we often find the byte pattern of `0x0000`{`Rung Signature`}{`Rung Size`}{`Opcode`} where `0x0000` represents the start of a rung. Since `Shade` cannot verify the two bytes of the `Rung Signature`, the system marks the byte sequence from `0x0000` to `Rung Size` as decompiled if `Rung Size` (two bytes) is valid ($\geq 8$) so long as the two bytes of the `Opcode` is in the mapping table. Note that `Shade` will not mark the `Opcode` bytes as decompiled until it first verifies its operand part (e.g., operand size, the bit offset).

Interestingly, this indicates that this byte pattern, and potentially others, can be found in non-code data as well (although it is not often across all samples, it is non-negligble), since the byte sequence of `0x0000` commonly appears in non-code data. Beyond appearing in non-code data, we also find that the valid range of the rung size in this instance is significantly wide. On the contrary, the best n-gram feature for the MicroLogix 1400 is an 8-gram. This feature type involves a relatively large information space ($256^8$), indicating it would be less likely that an arbitrary 8-byte sequence in non-code packets happens to be a member of the 8-gram bloom filter.

We found another key insight regarding the optimal size of n-gram features. Specifically, we find a different n-gram optimal size for each PLC (i.e., 4-gram for Modicon M221 and 8-gram for MicroLogix 1400). As the size of n-gram increases, we see little improvement in detection rate at small n-gram sizes. Critically however, as the size approaches certain size boundary, we see a steep rise in the successful detection rate. Gradually, the detection rate declines as the n-gram size rises beyond this boundary. Related to this discovery, we note that the *#Ngram* and *LNgram* features show similar detection rates when the size of the n-gram is the same, implying that the size of an n-gram is a more important factor than whether or not the pre-defined n-grams are *continuously* present.

### 6.2 Classification Results

Based on the selected features, we employed two classic machine learning algorithms, the Gaussian Naive Bayes and a Support Vector Machine (SVM) with robust, non-linear RBF kernel. We utilize these models to generate two independent sets of classification results and then compare the detection performance of each. Figure 8 highlights the ROC curves with an FPR limit of 1%. For the training datasets, it represent a mean ROC curve generated by 10-fold cross-

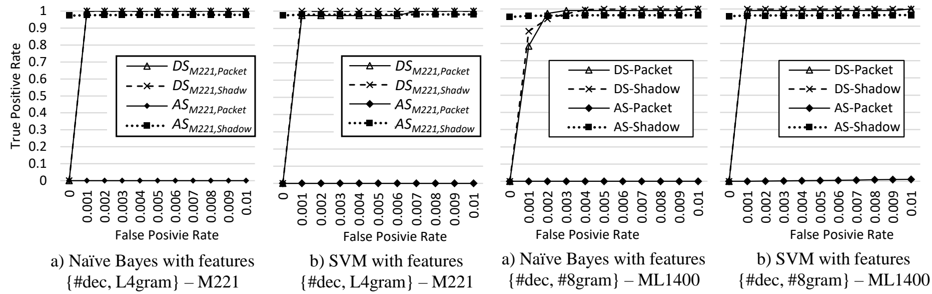



**Fig. 8.** ROC curves with 1% of FPR limit

validation. For both PLCs, packet-basis DPI shows near 0% of detection rate against the evasion attacks regardless of the classification algorithms. By contrast, `Shade` shows above 96% of detection rate in all cases with 0.1% of the FPR limit. As displayed in Figure 8, the types of classification algorithms do not significantly affect the detection rate, but the **use of shadow memory has created a critical difference**. Notably, our findings validate empirically that the evasion attacks from [27] are in-fact effective against traditional packet-basis DPI, *yet* `Shade` can successfully detect these attacks.

We further analyze the detection results to calculate the detection rates of each *attack instance*. We consider each control logic program as a unique attack instance. For example, we assume that there are 29 attack instances against the Modicon M221 PLC since there are 29 unique control logic programs in the attack datasets. Likewise, there are 75 attack instances against the MicroLogix 1400 PLC. We mark each attack instance as detected if we detect one of its code packets. Note that a control logic program will not be successfully executed in a PLC if one of its code fragment is missing.

Table 3 and 4 show the detection rates of attack instance at 0% FPR. These tables also show how the detection rate varies with differing feature sets. We used only the SVM classification models for this analysis. For every feature set for Modicon M221, `Shade` perfoms with a 100% detection rate of attack instances while packet-basis DPI shows 0% detection rate, due to the evasion techniques of the attacks described in the earlier background. For packet detection instead of instance detection, the feature set of {*#def*, *L4gram*} performs best with a detection rate of 97.52%. We find similar results for the MicroLogix 1400 (refer to Table 4), indicating `Shade` and the evaluation approach we employed apply to two distinct, yet practically deployed PLCs on the market. The attack instance detection rates reach 100% in `Shade` while the packet detection rates are slightly different depending on feature set. We find the best packet detection rate of 95.68% with the feature set of {*#def*, *#8gram*} using `Shade`. As found with the other PLC, once again the packet-basis DPI fails to find any success, at 0% detection rates on both attack packets and instances.

### 6.3 Scan Boundary Parameter and Performance Overhead

In the previous evaluation results, we set the scan boundary parameter $b$ to the maximum payload size of a PLC protocol, i.e., 236 for Modicon M221 and 80 for

**Table 3.** Detection rates at 0% FPR - M221

| Feature Set | Packet-basis DPI | | Shade | |
|---|---|---|---|---|
| | Attack packet | Attack instance | Attack packet | Attack instance |
| {#dec} | 0% (0/3865) | 0% (0/29) | 92.23% (3565/3865) | 100% (29/29) |
| {L4gram} | 0% (0/3865) | 0% (0/29) | 95.08% (3675/3865) | 100% (29/29) |
| {#dec,L4gram} | 0% (0/3865) | 0% (0/29) | 97.52% (3769/3865) | 100% (29/29) |

**Table 4.** Detection rate at 0% FPR - MicroLogix 1400

| Feature Set | Packet-basis DPI | | Shade | |
|---|---|---|---|---|
| | Attack packet | Attack instance | Attack packet | Attack instance |
| {#dec} | 0% (0/24866) | 0% (0/75) | 92.39% (22974/24866) | 100% (75/75) |
| {#8gram} | 0% (0/24866) | 0% (0/75) | 93.90% (23348/24866) | 100% (75/75) |
| {#dec,#8gram} | 0% (0/24866) | 0% (0/75) | 95.68% (23793/24866) | 100% (75/75) |

**Table 5.** Performance according to scan boundary $b$ (FPR: 0%) - M221

| | Shade | | | | | Packet DPI |
|---|---|---|---|---|---|---|
| $b$ (boundary) | 236 (Max) | 16 (4n) | 8 (2n) | 6 (1.5n) | 4 (n) | - |
| Attack Packet | 93.69% (3621) | 84.58% (3269) | 64.19% (2481) | 54.77% (2117) | 6.88% (266) | 0% (0) |
| Instance | 100% (29) | 100% (29) | 100% (29) | 100% (29) | 68.97% (20) | 0% (0) |
| Time (sec) | 16.82 | 13.29 | 13.27 | 13.01 | 12.9 | 12.75 |
| Overhead | 31.92% | 4.24% | 4.08% | 2.04% | 1.18% | - |

**Table 6.** Performance according to scan boundary $b$ (FPR: 0%) - MicroLogix 1400

| | Shade | | | | | Packet DPI |
|---|---|---|---|---|---|---|
| $b$ (boundary) | 80 (Max) | 32 (4n) | 16 (2n) | 12 (1.5n) | 8 (n) | - |
| Attack Packet | 94.25% (23437) | 92.76% (23065) | 91.17% (22670) | 87.71% (21810) | 29.30% (7286) | 0% (0) |
| Instance | 100% (75) | 100% (75) | 100% (75) | 100% (75) | 97.33% (73) | 0% (0) |
| Time (sec) | 421.2 | 416.9 | 415.6 | 414.6 | 414.0 | 410.8 |
| Overhead | 2.53% | 1.48% | 1.17% | 0.93% | 0.78% | - |

MicroLogix 1400. However, we believe it would also be interesting to examine how $b$ affects the detection rate and performance overhead. For this purpose, we examine varying configurations of $b$ between $n$ to the maximum payload size, where we set $n$ to the size of the selected feature of the n-gram for each PLC (i.e.. In our analysis, $n$ is 4 for the Modicon M221 and 8 for the MicroLogix 1400). In this analysis, we use the n-gram feature alone (without *#dec*) for each PLC, i.e., *L4gram* for Modicon M221 and *#8gram* for MicroLogix 1400. This was conducted with Gaussian Naive Bayes classifiers.

Our findings are highlighted in Table 5. Specifically, the results show detection rates at 0% of FPR and time overhead for Modicon M221. We average the time from 10 individual executions to ensure consistency. The baseline of overhead computation is the packet-basis DPI which does not employ shadow memory. When $b$ is equal to or greater than 6 ($1.5n$), `Shade` detects all the attack instances while the detection rates of attack packets are different depending on the configured value for $b$. When $b$ is 236 (the maximum payload size for Modicon M221), the approach taken by `Shade` shows the best packet detection rate of 93.69%, with a temporal cost of 31.92% performance overhead.. However, as we discussed earlier, this is enough for a defender to prevent remote control-logic injection attacks if she can detect all the attack *instances* over the network, since an attacker's control logic program cannot be executed successfully in a PLC if a missing code fragment exists. Therefore, the optimal configuration of $b$ can be

6 with which Shade detects all the attack instances with only 2.04% of overhead. As for memory, Shade allocated 196KB for shadow memory throughout the execution.

In the case of the MicroLogix 1400 (refer to Table 6), we set $n$ to 8 because *#8gram* performed best among n-gram features. When $b$ is equal to or greater than 12 ($1.5n$), Shade shows 100% of attack *instance* detection rate. The highest attack *packet* detection rate is 94.25% when $b$ is 80 (the maximum payload size for the PCCC protocol) with 2.53% of time overhead. In the MicroLogix 1400 case, the optimal configuration of $b$ can be 12 ($1.5n$), where we find Shade performs at 100% of attack instance detection rate with only 0.93% of overhead. We find a slightly higher total allocated memory size for the shadow memory in the case of the MicroLogix PLC, at 6.16MB throughout the execution.

Performance overhead is an important factor especially in the ICS domain where prompt responses on the attacks attempt to manipulate physical process is critical. Furthermore, when Shade is deployed in a network-based IPS, rapid decision making is necessary to minimize the delay on the communication over the network. Note that for some types of ICS domain specific messages, their transfer time requirements could be very tight. For instance, the IEC 61850-5 standard specifies performance requirement for different types of messages for electrical substations [12], and it requires that the transfer time must not exceed 3 ms for the messages of trips and blocking.

### 6.4 Discussion on the Evaluation Results

The evaluation results clearly signal that the *shadow memory* approach via Shade performs signficiantly well for detection of *stealthy* control logic injection attacks, whereas the traditional packet-basis DPI fails in all cases. These findings also suggest detection models rely most strongly on the features of *decompiled* bytes and a certain size of *n-gram* in the ICS domain, specifically control logic. Generally, generating a decompiler for the control logic code of a PLC may require a painstaking set of reverse engineering tasks. On the other hand, n-gram features *require no semantic knowledge of control logic code*, thus these can be generally applied to other PLC types. In an environment such as ICS, where the operating systems and instruction architectures vary significantly more than the standard Windows, Mac, or Linux-based distros in traditional IT, this critical device-independent semantic analysis may be a promising, less costly path forward both for operator deployment and future academic research. In a similar manner, the 0% FPR assuages concerns of false alarms in practice, a far worse situation in the ICS domain over traditional false alarms for malware detection or network intrusion detection in IT environments. Wasting an analyst's time in the IT domain may only result in a user's computer being confiscated or re-imaged for compliance reasons. However, shutting down a water treatment plant's PLC network or nuclear reactor control system and then later discovering it was a false alarm can cost both significant financial resources as well as put human lives in danger.

We conjecture that the reason for the 0% FPR is because our classification approach is a type of *misuse detection*[5] rather than an anomaly detection. However, unlike common misuse detection scenarios in the IT domain, our approach should not be limited to detecting existing control logic injection attacks. In the case when an attacker wants to download logic to a PLC, the attackers logic code necessarily shares some common characteristics of control logic code (e.g., opcode, rung structure, etc.).

**Deployment scenario.** `Shade` needs to see the network traffic between an engineering workstation and PLCs. In most cases, the engineering workstation is at a control center network based on TCP/IP and Ethernet. Therefore, `Shade` can utilize network taps at the control center network to monitor the network traffic between the engineering workstation and PLCs. On the other hand, `Shade` also can be deployed in a network-based IPS or industrial firewall [28].

## 7 Related Work

**Control Logic Injection Attacks.** Stuxnet [8] best represents real-world cases of control logic attacks which target specific PLC types (Siemens S7-300) and its engineering software (Siemens SIMATIC STEP 7). Stuxnet sabotaged Iran's nuclear facilities by infecting engineering software[6] and then injecting malicious control logic to target PLCs using the infected engineering software.

Senthivel *et al.* [24, 23] presents three control logic injection attack scenarios referred to as denial of engineering operations (DEO) attacks where an attacker can interfere with the normal engineering operation of downloading/uploading of PLC control logic. In DEO I, an attacker in a man-in-the-middle (MITM) position between a target PLC and its engineering software injects malicious control logic to the PLC and replaces it with normal (original) control logic to deceive the engineering software when uploading operation is requested. DEO II is similar to DEO I except that it uploads malformed control logic instead of the original control logic to crash the engineering software. DEO III does not require MITM positioning, in which the attacker simply injects specially crafted malformed control logic to the target PLC. The malicious control logic is crafted in a way that it can be run in the PLC successfully, but the engineering software cannot decompile the control logic.

**Network Intrusion Detection for PLCs.** Digital Bond's Snort rules [25] represent a classic approach of NIDS for ICS. Their primary purpose is to detect violation of protocol specification or unauthorized usage of the protocol's function code (e.g. write request, cold restart, and disable unsolicited responses). There are also rules to detect control logic downloading to a PLC. However, these rules only inspect the PLC protocol header, allowing easy evasion by manipulating the header value as described in [27].

[5] We extract features based on the properties of control logic code and decide code packets as malicious in our evaluation scenario

[6] Stuxnet replaces original s7otbxdx.dll of STEP 7 with its own version to intercept communication between STEP 7 and S7-300 PLC

Hadiosmanovic *et al.* [15] present a semantic-oriented NIDS to detect abnormal behavior of a physical process. They infer PLC variable types based on the degree of variability: control variables appear constant, reporting and state variables can be mapped to a discrete set of values, and measurement variables are usually continuously changing. They build behavioral models for each variable to detect a significant deviation between the model and observed series of data. They apply autoregressive modeling to model measurement variables (high variability) and derive a set of expected values (as a white-list) for other types of variables (medium or low variability). Since their approach focuses on data instead of code, it can only detect post-attack effects after the attacker's control logic code is injected into the target PLC, which may reflected in the data blocks.

**Deep Packet Inspection.** PAYL [2] is a payload anomaly detection system based on a 1-gram analysis. For each packet payload, it counts the relative frequency of each 1-gram, thereby each packet payload is embedded in a 256-dimensional feature space. Based on this, it generates payload models per host, flow direction (inbound/outbound), service port, and payload length, which are represented by mean and standard deviation of each feature ($0 \sim 255$). Then, it uses Mahalanobis distance in the detection phase to measure the distance between payload under test and corresponding payload model. However, as pointed out in [5], it can be evaded by mimicry attacks since its analyzed data unit is only 1 byte (1-gram).

Anagram [4] uses higher-order n-grams ($n>1$) to be resistant to against mimicry attacks. The payload modeling technique used in PAYL is not suitable for higher-order n-grams because the feature space grows exponentially as $n$ increases ($256^{n}$). Therefore, they utilize bloom filters to extract n-gram features, as we also did in this paper. They record each n-gram of the payload in the training dataset using bloom filters. Then, in the detection phase, a payload is scored by counting the number of n-grams which are not a member of the bloom filter. However, this technique alone is not suitable when evasion attacks are involved, as demonstrated in [27]. For example, in the Fragmentation and Noise Padding attack, attacker's code fragment in each packet can be very small (even one or two bytes) to which a large amount of non-code padding is appended, making it difficult for packet-basis DPI to detect the attack packets.

## 8 Conclusion

In this paper, we introduced a novel deep packet inspection (DPI) technique, `Shade`, based on shadow memory to detect control logic in ICS network traffic against the evasion attacks presented in recent literature. As a part of developing the proposed DPI technique, we analyzed five different types of features (42 unique features overall) at different semantic levels including decompilation, rung and opcode identification, n-gram, and entropy. We implemented and evaluated our approach on real-world PLCs from two different vendors. Our evaluation results show that the evasion attacks can subvert a traditional packet-basis DPI while `Shade` can detect the attack instances with nearly 100% accuracy at a

0% false positive rate. We also show that the performance overhead of shadow memory is only about 2% when using an optimal scan boundary parameter.

## References


[1] Toth, Thomas and Kruegel, Christopher: Accurate buffer overflow detection via abstract payload execution. pp. 274–291 (2002)
[2] Wang, Ke and Stolfo, Salvatore J: Anomalous payload-based network intrusion detection. vol. 3224. Springer (2004)
[3] Chinchani, Ramkumar and Van Den Berg, Eric: A fast static analysis approach to detect exploit code inside network flows. vol. pp., 284–308. Springer (2005)
[4] Wang, Ke and Parekh, Janak J. and Stolfo, Salvatore J.: Anagram: A content anomaly detector resistant to mimicry attack. In: Proceeding of the 9th International Conference on Recent Advances in Intrusion Detection (RAID) (2006)
[5] Fogla, Prahlad and Sharif, Monirul and Perdisci, Roberto and Kolesnikov, Oleg and Lee, Wenke: Polymorphic blending attacks. In: Proceedings of the 15th Conference on USENIX Security Symposium (2006)
[6] Nethercote, Nicholas and Seward, Julian: Valgrind: A framework for heavyweight dynamic binary instrumentation. pp. 89–100 (2007)
[7] I. N. Fovino and A. Carcano and T. D. L. Murel and A. Trombetta and M. Masera: Modbus/dnp3 state-based intrusion detection system. pp. 729–736 (2010)
[8] Falliere, Nicolas and Murchu, Liam O and Chien, Eric: W32. stuxnet dossier. White paper, Symantec Corp., Security Response 5(6), 29 (2011)
[9] Serebryany, Konstantin and Bruening, Derek and Potapenko, Alexander and Vyukov, Dmitry: Addresssanitizer: A fast address sanity checker. pp. 28–28 (2012)
[10] Irfan Ahmed and Sebastian Obermeier and Martin Naedele and Golden G. Richard III: SCADA systems: Challenges for forensic investigators. Computer 45(12), 44–51 (2012)
[11] IEC 61131-3 Ed. 3.0 b:2013, Programmable controllers - Part 3: Programming languages. Standard, International Electrotechnical Commission (2013)
[12] IEC 61850-5 Ed. 2.0:2013, Communication Networks and Systems for Power Utility Automation - Part 5: Communication requirements for functions and device models. Standard, International Electrotechnical Commission (2013)
[13] Lee, Robert M and Assante, Michael J and Conway, Tim: German steel mill cyber attack. Tech. rep., SANS (2014)
[14] ICS Focused Malware. https://ics-cert.us-cert.gov/advisories/ICSA-14-178-01 (2014), [Online; accessed 03-June-2018]
[15] Hadžiosmanović, Dina and Sommer, Robin and Zambon, Emmanuele and Hartel, Pieter H.: Through the eye of the plc: Semantic security monitoring for industrial processes. In: Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC) (2014)
[16] McLaughlin, Stephen E and Zonouz, Saman A and Pohly, Devin J and McDaniel, Patrick D: A trusted safety verifier for process controller code. In: Proceeding of the 21st Network and Distributed System Security Symposium (NDSS) (2014)
[17] Cyber-Attack Against Ukrainian Critical Infrastructure. https://ics-cert.us-cert.gov/alerts/IR-ALERT-H-16-056-01 (2016), [Online; accessed 03-June-2018]
[18] ICS-CERT Annual Vulnerability Coordination Report. Report, National Cybersecurity and Communications Integration Center (2016)
[19] Ahmed, Irfan and Roussev, Vassil and Johnson, William and Senthivel, Saranyan and Sudhakaran, Sneha: A scada system testbed for cybersecurity and forensic research and pedagogy. In: Proceedings of the 2nd Annual Industrial Control System Security Workshop (ICSS) (2016)
[20] CRASHOVERRIDE Malware. https://ics-cert.us-cert.gov/alerts/ICS-ALERT-17-206-01 (2017), [Online; accessed 03-June-2018]
[21] Cinelli, Mattia and Sun, Yuxin and Best, Katharine and Heather, James M and Reich-Zeliger, Shlomit and Shifrut, Eric and Friedman, Nir and Shawe-Taylor, John and Chain, Benny: Feature selection using a one dimensional naïve bayes classifier increases the accuracy of support vector machine classification of cdr3 repertoires. Bioinformatics 33(7), 951–955 (2017)
[22] Irfan Ahmed and Sebastian Obermeier and Sneha Sudhakaran and Vassil Roussev: Programmable logic controller forensics. IEEE Security Privacy 15(6), 18–24 (November 2017)
[23] Senthivel, Saranyan and Ahmed, Irfan and Roussev, Vassil: Scada network forensics of the pccc protocol. Digital Investigation 22, S57–S65 (2017)
[24] Senthivel, Saranyan and Dhungana, Shrey and Yoo, Hyunguk and Ahmed, Irfan and Roussev, Vassil: Denial of engineering operations attacks in industrial control systems. In: Proceeding of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY) (2018)
[25] Digital Bond's IDS/IPS rules for ICS. https://github.com/digitalbond/Quickdraw-Snort (2018), [Online; accessed 19-July-2018]
[26] Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed: Clik on plcs! attacking control logic with decompilation and virtual plc. In: Proceeding of the 2019 NDSS Workshop on Binary Analysis Research (BAR) (2019)
[27] Hyunguk Yoo and Irfan Ahmed: Control logic injection attacks on industrial control systems. In: 34th IFIP International Conference on Information Security and Privacy Protection (2019)
[28] Tofino Xenon Security Appliance. https://www.tofinosecurity.com/products/tofino-xenon-security-appliance (2019), [Online; accessed 17-April-2019]