PLC Guard: A Practical Defense against Attacks on Cyber-Physical Systems

Jan-Ole Malchow*, Daniel Marzin*, Johannes Klick*, Robert Kovacs†, Volker Roth*
*Secure Identity Research Group
Freie Universität Berlin
<firstname>.<lastname>@fu-berlin.de
†Hasso-Plattner-Institut für Softwaresystemtechnik GmbH
robert.kovacs@hpi.com

***Abstract*—Modern societies critically depend on cyber-physical systems that control most production processes and utility distribution networks. Unfortunately, many of these systems are vulnerable to attacks, particularly advanced ones. While researchers are investigating sophisticated techniques in order to counter these risks, there is a need for solutions that are practical and readily deployable. In this paper, we adapt the classic ACCAT Guard concept to the protection of programmable logic controllers (PLCs), which are an essential ingredient of existing cyber-physical systems. A PLC Guard intercepts traffic between a, potentially compromised, engineering workstation and a PLC. Whenever code is transferred to a PLC, the guard intercepts the transfer and gives the engineer an opportunity to compare that code with a previous version. The guard supports the comparison through various levels of graphical abstraction and summarization. By operating a simple and familiar interface, engineers can approve or reject the transfer using a trusted device that is significantly harder to subvert by attackers. We developed a PLC Guard prototype in order to reify our ideas on how it should be designed. In this paper, we describe the guard's design and its implementation. In order to arrive at realistic PLC code examples, we implemented a miniature packaging plant as well as attacks on it.**

I. Introduction

Mostly unnoticed by the populations of modern societies, so-called cyber physical systems control and automate most of their vital production and utility distribution processes. Failures of individual systems can bring down entire production lines or energy distribution networks. Additionally, such failures can cause ripple effects that spread to neighboring dependent systems. It is therefore concerning that security awareness is still lacking in many industrial segments even though such systems are increasingly connected to the Internet in order to facilitate remote management and reporting. This exposes these systems to espionage or sabotage by state actors, criminals, anarchists and terrorists. Even when airgapped, such systems are not immune to attacks. The Stuxnet incident [1] has brought this risk to the attention of governments and the public.

Stuxnet targeted the control level of the automation pyramid, which consists of so-called programmable logic controllers (PLCs). In order to achieve its goal, Stuxnet infected Windows computers with Siemens' STEP 7 development software installed. Stuxnet uploaded a malicious PLC program to the PLC that took over motor control from the original program.

Fig. 1: Shows our custom-designed box housing the PLC Guard prototype.

Preventing attacks of this kind is challenging because industrial software development tools run on widespread commodity operating systems that have always been a primary target for malware writers. The situation is exacerbated by the complexity of configuring such a computer system and the required applications in ways that meet security requirements and usability requirements in a production environment. In a production environment, it is typically more important to "get things done" rather than to delay activities because of vague security concerns that are unfounded, most of the time. Towards a solution of that problem, McLaughlin et al. [2] have proposed a trusted device, the Trusted Safety Verifier (TSV), which verifies whether a given PLC program meets a given safety property. The TSV suffers from an exponential runtime complexity and the graphs in the aforementioned publication confirm this expectation. Nonetheless, McLaughlin et al. [2] argue the overhead is manageable for real-world PLC code verification based on experimentation with six example control tasks.

We were interested to validate this hypothesis and to create a realistic scenario suitable to experiment with PLC protection mechanisms and attacks. With this in mind, we designed, built and programmed a miniature packaging plant. The plant is based on industrial components that we control with a single Siemens S7-313C PLC. Our experiences, which we share in §VIII-A, do not support the hypothesis that automated safety verification is a general solution. In order to cover the range of automation systems that exist, we need to offer more pragmatic protection mechanisms. Towards a more pragmatic approach, we implemented a PLC Guard, which is shown in Figure 1. Guards are a classic concept [3], [4] originally developed at the *Advanced Command and Control Architectural Testbed*

(ACCAT). The ACCAT Guard was a trusted interactive device designed to support the sanitization and downgrading of "high" data under the control of human operators and a Security Watch Officer. The function of the Officer was to review and approve (or deny) downgrades. Once downgraded, data could be output to a "low" network interface. Likewise, our PLC Guard is a trusted device that transparently intercepts transmissions of PLC code from the engineering workstation to the PLC. It decodes the MC 7 code that is the basis of all Siemens PLC programs and contrasts it with the previous version of the code on a trusted display. An engineer who fulfills the role of the Security Watch Officer approves or denies the code upload by a physical interaction with the guard. If he approves then the guard loads the PLC program onto the PLC. Network traffic other than traffic related to PLC code transmissions is forwarded transparently. Hence, whereas the ACCAT Guard enforced a multi-level confidentiality policy, our guard enforces an integrity policy. The guard pattern is conceptually straightforward such as most trusted component patterns like firewalls and intrusion detection systems. What matters is how a pattern is reduced to practice. An objection against a PLC guard might be that comparing PLC code is notoriously difficult and the tediousness of the task will prompt operators to simply wave uploads through. Indeed, we adopt this as our "null" hypothesis and seek to accept an alternative hypothesis instead, which is that the comparison can be made sufficiently straightforward so that dilligent engineers can perform it without much cognitive strain. We seek to achieve this by leveraging the structure of PLC code and meta-properties that we can extract from it. Based on this information, we design a review process such that individual decisions progress from automatic checks to semi-automatic checks and from syntactic checks to semantic checks. This allows to weed out large classes of malware with little cognitive overhead. Comparison of small amounts of actual code is only the last step in a process that reduces risks to all but the most constraint malware. We discuss this process and the reasoning behind it at length in the paper.

In §II we describe our threat model, in §III describes our design, §IV describes the review process in detail, §VI gives details on our implementation, §VII summarizes our evaluation, in §VIII we discuss related work, and §IX concludes our paper.

II. Threat Model

We assume that the adversary controls an engineering workstation through a backdoor or a malware that runs with administrative privileges on the workstation. The workstation connects to a PLC over an IP network. The PLC is the adversary's target and his goal is to run malicious code on the PLC. The adversary may modify any PLC code that the engineer writes on the workstation and that the PLC downloads from the workstation. However, we assume that the adversary cannot connect to the PLC in ways other than using the workstation's outgoing network interface. The Stuxnet scenario falls squarely into our threat model. We assume that the engineer is honest.

III. Guard Design

The guard acts as a transparent proxy between a workstation and a PLC. Transparency is required so that existing network configurations do not break. For this reason, the guard forwards all traffic except *S7 communication* download requests. If it detects a download request it does not forward the request but takes over and executes the download command in order to fetch the code from the EWS. The downloaded code is passed to a MC 7 disassembler and the output is stored for review. If a reviewer approves the code for download then the guard poses as the EWS, sends download commands to the target PLC and answers its download requests. The guard is meant to be deployed close to the EWS that engineers use to configure and program PLCs. This requirement is rooted in the fact that engineers need to interact with the guard in order to complete code transfers to a PLC. The guard acts as a reference monitor [5] with respect to the PLC code transfers between the EWS and a PLC. Hence, it should exhibit similar properties provided that the underlying operating system and its basic services are secure. In other words, the guard: 1) must be tamperproof, 2) must always be invoked, and 3) must be small enough to be subject to analysis and tests to assure that it is correct. As any trusted device, the guard's design and implementation should conform to good design principles for secure systems [6]. In what follows, we sketch our interpretation of some of these requirements and properties in the context of a PLC Guard device.

A. Tamperproofness

Our guard implementation is a prototype and as such it is conceptually tamperproof and hacker-proof but not realistically so. However, systems such as Honeywell's SCOMP [7] and NSA's Blacker [8] have demonstrated that small systems can be engineered with a high assurance level. If any physical tampering is detected, the guard must move to a secure state, while possibly raising an alarm.

B. Mandatory Invocation

We designed our guard with two network interfaces so that it can be plugged easily in between an EWS and its outgoing network connection. Recall that the guard is meant to protect against an infected workstation and not against an attacker elsewhere in the network. As is the case with any trusted network component, if the network topology allows bypassing it then it cannot be effective. Once it is operative, the guard acts as a transparent proxy. It analyzes all network traffic between the EWS and PLCs and intercepts specific *S7 communication* requests while forwarding all other network traffic. This is necessary in order to fit into existing industrial Ethernets without requiring changes. The guard never allows a PLC code transfer without physical interaction with an engineer.

C. Minimal Design

In order to minimize the design of the guard we followed the principle of "one tool for one task." Towards this end, we split the guard into an enforcement component and a separate review device. This leads to smaller implementations, which is a desirable property for security-critical software. The design is such that information only flows from the enforcement component to the to the review device.

IV. PLC Program Review Process

During the program review phase, the engineer is tasked to decide whether differences between two versions of a PLC reflect the changes he made from the previous version of his code to the most recent one. Without decision aids, this task is likely to be very difficult. Fortunately, PLC code is highly structured. We argue that the decision task can be supported efficiently and effectively for PLC programs. In our argument, we first step through an ordered sequence of program checks and derive constraints from them. The order of checks is such that automatic checks come before semi-automatic and manual checks and syntactic checks come before semantic checks. This leads to a sequence that minimizes expected cognitive effort. Each constraint we derive excludes certain types of malicious behaviors and imposes limits on malicious code, that is, the behavior is prevented or a diligent engineer will detect the malcode. We conclude this section with a discussion of specific attack patterns of interest and residual risks. Our goal is not to provide perfect protection against all conceivable attacks but to raise the risks and costs of attacks to make them uneconomical or unlikely to inflict lasting damage. Most checks are supported by graphical visualizations that summarize signals of malcode so that an engineer can detect them easily and efficiently at a high level of abstraction. Here, *easy* means that a signal manifests as a graphical attribute (for example, color) that can be interpreted at a glance and without having to remember the details of a PLC program. In what follows, we first give an overview over the types of visualizations the guard supports, followed by a derivation of constraints.

A. Visualizations

Software visualization has evolved significantly in recent years and has proven to be a valuable tool to understand software. For example, graphs have been used to examine the evolution of sourcecode [9], [10] and to trace bugs [11]. In general, visualizations work best when tailored to a particular problem domain [12]. PLC programs evolved from wired circuits and their internal structure still reflects this heritage. For example, instructions correspond to the functions of switching elements whereas inputs and outputs correspond to terminals in a circuit. The visual programming languages *Functional Block Diagram* (FBD) and *Ladder Logic* (LAD) emphasize this structure. Consequently, the effects of changes to a PLC program on its MC7 representation are highly localized. Furthermore, changes may only affect the "wiring" of blocks or they may replace program segments with other ones. PLC programmers tend to have an electrical engineering background rather than a computer science background. A visualization of actual source code may put them at a disadvantage compared to software engineers. For example, McKeithen et al. [13] already found that expert programmers build internal structures that help them recall programs better than novice programmers. This is consistent with findings in other areas that studied expert versus novice performance, for example, playing chess, go, bridge, music, electronics and physics. However, an electrical engineer will remember whether he modified wirings or merely calibrated switching elements, for example. This immediately leads to graphs as a representation of PLC programs. The guard supports graph displays at two levels of detail. In order to distinguish between them let *block* denote an FB, FC, OB, SFB, or SFC. A block may contain STL code with an internal branching structure and calls to other blocks. A block may be further subdivided into what is known as *basic blocks* in the compiler literature, that is, a sequence of instructions such that one instruction always executes before all instructions in later positions and no other instruction executes between two instructions in the sequence. The first level displays PLC programs at the level of blocks. We refer to this as the *inter-block* structure. The second level displays a block at the level of basic blocks. We refer to this as the *intra-block* structure. Our inter-block display places inputs, outputs, memory words (MW) and data blocks (DB) at well-defined positions so that engineers can fixate them without effort. Inputs are at the top and outputs are at the bottom. DB are to the left and MW are to the right. Edges represent relations between different elements. The state of a relation: unchanged, new, deleted, modified and its multiplicity is coded by means of visual representations like color or thickness. Positioning the mouse pointer on top of an element brings all adjacent edges to the front. Clicking on a block opens a view for intra-block analysis, which shows the differences of this block and its previous version. We give an example of the graphical representation in Figure 2. The visualization also shows statistics and meta-information on the program which play a role in the reviewing steps we explain next, for example, block counts and lines of code. Our intra-block display shows basic blocks connected by edges that indicate control-flows between the basic blocks. Each basic block bears a label. Labels are used in branches and calls to indicate the target of the branch or call (see Fig. 2).

B. Automatic Protections

Window of opportunity: At the most fundamental level, the PLC Guard limits attacks to a PLC to the time of software maintenance. While a compromised EWS may upload malcode to a PLC at any time, this is not possible if the guard is present. Uploads occur only in the case of a trusted physical interaction by an engineer that is not bypassable in software. This is already a significant step forward towards protecting a PLC because limiting maintenance overhead is a priority in industrial process engineering and optimization.

Malcode constraint 1. *Manipulation occurs only at the time of maintenance.*

Calling conventions: TIA wraps calls in BLD commands that are not strictly necessary for a syntactically correct and functioning MC7 program. However, this allows TIA to transform calls back into its higher-level representation. The PLC guard verifies these calling conventions and hence malcode must comply with them. As a consequence, deleting or adding a jump requires a change of at least five lines of code instead of one, which is more noticeable.

Dead code detection: The guard performs a fall-through disassembly of MC7 code and verifies that unconditional and conditional branches only branch to valid instructions. This check is straightforward compared to other low-level architectures because MC7 supports only direct offsets in branches, indirect branches are not supported. If code exists that is not potentially reachable then the guard rejects the program

because PLC programs do not contain dead code and dead code is thus indicative of attempted manipulation.

Malcode constraint 2. *All code is subject to analysis and all code is syntactically correct with regard to its control-flow structure.*

System functions (FC) consistency: TIA transfers system function blocks to the PLC only if the PLC program actually requires them. The code of system function blocks should not change during maintenance. The PLC guard keeps fingerprints (cryptographic hashes) of all system function blocks and refuses programs whose function blocks do not match the known fingerprints. The (rare) case of system function updates requires an authenticated guard update.

Malcode constraint 3. *Manipulation occurs only in user generated code.*

C. Inter-Block Checks

Changes in PLC code exhibit strong locality due to the absence of inter-procedural or even intra-procedural optimizations. Predictable program behavior and backtranslation are more desirable in industrial programs than performance optimizations. The benefit is that changing a source code fragment does not affect MC7 code other than what corresponds directly to that fragment. Along the same lines, relationships among blocks and between blocks and I/O are generally stable. If they change then because the engineer reprogrammed them explicitly. This enables a number of simple yet effective checks. *Block state*: All blocks are represented as rounded rectangles. The state of each block is color-coded. Possible states are: *unmodified* and *modified.* The modified state has three sub-states: *new, deleted* and *changed.* The color-coding renders any manipulations immediately obvious that are not due to maintenance.

Malcode constraint 4. *Block manipulation is detected easily unless it is limited to blocks that changed due to maintenance.*

Block relations: The relationships of blocks, input, output, data and memory are represented as arrows, that is, directed edges. The thickness of an arrow encodes the multipliciy of the relationship it symbolizes. The color of an arrow encodes the state of the relatonships. Possible states are: *unmodified* and *modified.* The modified state has three sub-states: *new, changed* and *deleted.* The relations of blocks that were not modified during maintenance do not change. Since arrows represent inter-block control flow, surreptitious changes to a program's control flow or changes to I/O relationships become immediately obvious unless they are due to maintenance.

Malcode constraint 5. *Manipulation of control flow or I/O relationships is detected easily unless it is limited to relationships that changed due to maintenance.*

Lines of code: Our guard displays the STL lines of code measurements of the new code and its previous version. The number of source lines of code is proportional to the expected number of lines of STL code that the guard produces from intercepted MC7 code. This means that malcode cannot enlarge a PLC upload significantly beyond what is proportional to the original code without raising suspicion. While this metric leaves significant room to interpretation and error it does constrain malware that is intended to remain stealthy in complex control situations. Stuxnet is one such example and we discuss this further in Section V-C.

Malcode constraint 6. *Manipulated code and original code must have similar lengths.*

A special type of inter-block checks are function checks. Advanced functions such as network communication cannot be implemented directly in STL or any higher-level language. Instead, so-called FC blocks provide these functions. These blocks are uploaded to a PLC only if the PLC program imports them. The role of FC blocks may be compared to the role that native libraries play for higher-level programming languages such as Java. Function blocks are grouped based on a common theme such as math or network communication. *Function types used:* Our guard provides a list of all the function groups that a PLC program and its previous version import. Each group has a color-coded state: *unmodified* and *modified.* The modified state has three sub-states: *new, changed* and *deleted.* The state *changed* symbolizes that the new PLC code uses different functions of that group compared to the previous version of the code. The state *new* symbolizes that the new code imports functions from a group that has not been imported by the previous version of the code. The state *deleted* symbolizes the removal of all function calls of that group. What this means is that malcode that imports networking functions will attract attention and scrutiny immediately if the original code did not import the networking group.

Malcode constraint 7. *Manipulation is detected easily unless its functions are limited to the function groups imported by the original program.*

Functions used: Our guard also provides the list of functions used by the PLC program and its previous version. Each displayed function has a color-coded state, similar to what we have described before. The state *changed* means that the number of calls to the function has changed from one program version to the next. In that state, the function display includes the difference of the number of calls. This yields a refinement of the previous constraint.

Malcode constraint 8. *Manipulation is detected easily unless its functions are limited to the functions used by the original program.*

D. Intra-Block Checks

If inter-block checks do not indicate risks but the production environment requires a high degree of assurance then engineers can inspect the intra-block differences of PLC code. Two types of displays exist. The first type is the visualization of intra-block control flow we introduced before in Section IV-A. As in our inter-block visualization we highlight structural changes by means of color-coding. The second type is a side-by-side presentation of two versions of a basic block, with highlighting that allows engineers to scrutinize the differences more easily. Program comprehension can be aided further by various

source code presenting techniques. For example, Norcio [14] investigated the role of indenting on program comprehension. Miara et al. [15] found that small amounts of indentation work best, that is, indentation by 2-4 characters. Rambally [16] found that color-coding can help program comprehension. Raymond and Weimer [17] found that blank lines may aid local judgements of readability more than comments. We certainly cannot implement all these ideas in our research prototype but it is important to note that a body of knowledge exists that can be applied in order to make the engineer's task easier. Additionally, Buse and Weimar [18] proposed program summarization as a means to aid the understanding of source code differences. Given the variability of human behavior it is difficult to derive a precise constraint from basic block checks. What is clear is that the adversary is in the difficult position of having to guess how the engineer will behave and perform when the attack is under way. Even if the engineer only scrutinizes a small sample of differences, with some probability it is a difference that uncovers the attack, and the probability of choosing such a difference depends on the amount of maintenance changes. If an attack is detected then the target is warned and future attacks will become significantly harder.

Malcode constraint 9. *Manipulation of program code is limited to changes that are easily overlooked upon inspection of program code.*

E. Discussion

The malcode constraints we have established limit the amount of malcode, the functions the malcode can implement, the code locations in which malcode can be placed and when and how often an upload of malcode can occur even before an engineer is tasked to look at code differences. Even if an engineer inspects only a portion of the changes there remains a non-zero probability that the engineer will inspect those changes that are indicative of an attack. It is probably fair to say that these constraints already constitute significant limitations on the adversary and the attack. Attacks now require much more careful planning and orchestration and have an increased risk of detection.

V. Specific Attack Strategies

In this section, we discuss specific attack strategies ranging from low complexity and sophistication to high sophistication. For example, Stuxnet represents a complex and highly sophisticated attack because it attempted to conceal the effects of its attack and its presence from the operators of the enrichment facility it targeted. This required a manipulation of what human machine interfaces (HMI) displayed about the status of the enrichment process which, in turn, necessitated a period of data collection on the PLC. At a more abstract level, sophisticated malcode needs to perform the following tasks: 1) collect information about its environment 2) compute a trigger function that starts/pauses its activity 3) subvert normal operation in a concealed fashion 4) falsify information sent to human machine interfaces.

The size of a malcode necessarily reflects the complexity and sophistication of the attack. Malcode must invest code lines into replicating genuine functions as Stuxnet did (Constraint 6), connect to functions of the genuine program (Constraint 5), or modify the genuine program in places that perhaps were not subject to maintenance because they worked as intended (Constraint 4). On the other end of the spectrum are unsophisticated attacks that trigger immediately and achieve as much damage as possible before the effects are noticed and remediation-measures are initiated.

A. Immediate Effect Attacks

A worst case scenario is a manipulation that requires the modification of just one line of code and yet inflicts significant damage. By Constraint 4 the manipulation must be in code that has changed during maintenance. Maintenance changes are subject to quality assurance testing. We distinguish two scenarios.

In the first scenario, the manipulation is in the main code path and therefore the effect manifests immediately. However, hardly any critical infrastructure operators and few industries introduce maintenance changes into operational systems without quality assurance testing. If the effect manifests immediately then the manipulation will be caught with high probability during testing. However, it is important that a lack of operational security does not subvert the protection offered by the guard. It must be assured that the version of the code that is loaded onto the production system is the version that was inspected using the guard. Otherwise, a compromised EWS may detect whether code is sent to a test system rather than a production system. The EWS then manipulates code only if it is sent to the production system. The risk of lacking operational security can be mediated with the guard by uploading tested code to the production environment from the guard and not from the EWS.

In the second scenario, the manipulation is off the main code path. For example, a manipulation may manifest only in an emergency situation and cause a failure so that the emergency situation is not effetively remedied. By our assumption that the corresponding code has changed during maintenance it must be assumed that this code will be subject to testing. Otherwise, the manipulation will stand out in the guard's display because it is clearly not part of the maintenance changes.

B. Incremental Attacks

Adversaries may introduce incremental changes in uploaded programs that do not take effect immediately but only after the intended number of changes have been introduced and a trigger condition has been detected. The actual payload would thus lie dormant until a final change renders it active. Again, by Constraint 4 each change must hide in maintenance changes. If the attack requires changes in code that is never changed during maintenance then the attack will never complete. By Constraint 1 we can can lower-bound the expected time to complete the manipulation by $t \cdot n/c$ where t is the average time between maintenance updates, n is the number of lines needed for the manipulation and c is the number of code line manipulations per maintenance cycle. The success probability can be upper-bounded by $(1-p_c)^{n/c}$ where p_c is the probability that an engineer detects the manipulation of c lines of code during maintenance. Keeping in mind that maintenance of PLC

code is a rare event once a process is running (ranging from a hand full of times per year to never. it is clear that the guard forces adversaries to: (i) perform attacks of low sophistication, that is, keep n small, (ii) take considerable risks, that is, increase c or (ii) be very patient. It is illustrative to keep in mind that Stuxnet contained more than 19,000 lines of code. Irrespective of how one tweaks the numbers it is difficult to argue that a comparable manipulation that takes less than years to upload will not leave a non-trivial footprint during the attack.

C. Stealthy Attacks

Note again that a PLC program is detached from the actual I/O by the scan cycle. This allows easy simulation of inputs and outputs. An HMI may read or write I/O bits or MW of a PLC at any time. Reading typically happens at a certain interval which is called the *pull cycle*. Writing to input bits of the PLC may occur irregularly, for example, if the HMI simulates the pressing of a physical button connected to an input line. The PLC program is oblivious to read or write access by an HMI. Consequently, malcode has two options to remain stealthy. First, it may set output bits to genuine-looking values just before an HMI reads them. This requires timing and knowledge of the pull cycle. Second, if the HMI reads MW then the malcode may simulate the genuine program while driving output independently. The situation is complicated further if the malcode must react to HMI input. For example, if an operator turns a centrifuge off and the centrifuge continues to run then this would immediately alert operators to software problems. A convincing simulation requires a good model of the genuine behavior that can be computed efficiently or a sufficiently large trace that the malcode can replay. In either case, the malcode needs additonal MW or DB for storage, which introduces changes to the inter-block connections in the guard's display and violates Constraint 5.

VI. Guard Implementation

In order to reify our design, we built and implemented a prototype PLC Guard. It consists of a off-the-shelf Raspberry Pi model B equipped with an additional network interface with a custom-designed and 3D printed enclosing. We implemented a graphical representation of structural code differences, which is depicted in Figure 2.

A. Networking

We analyzed all network layers in order to assure that protocols other than *S7 communication* do not interfere with it. Our configuration uses two Ethernet interfaces, eth0 and eth1. eth0 is the interface to the EWS, while eth1 connects to the network where the PLC is located. We activated IP forwarding so that all IP traffic is forwarded. Since we intercept certain packets encapsulated in TCP packets, keeping sequence numbers synchronized is an issue. We solved this by splitting TCP connections (on port 102) between the EWS and the PLC into two separate connections. Incoming packets from the EWS are forwarded to the guard internal IP for the specific PLC. The guard application listens on port 102 on both the PLC-specific internal IP and the IP of eth1. It analyzes incoming packets and forwards them if necessary. In this fashion, two separate TCP stacks exist and the kernel handles everything as is usual. In order to make the guard invisible to the EWS, a source NAT entry in the postrouting table restores the original IP address of the specific PLC.

B. Guard Application

We implemented the guard application, including its *ISO-TSAP* and *S7 communication*. The guard application implements the required parts of the *ISO-TSAP* and *S7 communication* protocols. In the case of *ISO-TSAP*, the basic handshake is required. For *S7 communication*, upload and download requests are required. The guard checks for each incoming S7 packet whether the function parameter (first parameter byte) is $0x1A$. If it is not then the packet is forwarded directly. Otherwise the guard answers the request without forwarding it. The guard passes intercepted code to a MC 7 disassembler that we implemented. The development of the MC 7 disassembler required a semi-automatic process to extract the MC 7 code corresponding to each *STL* command from the TIA portal, since we only had the STL language description to work with. With the extracted information we compiled a lookup table for our disassembler. The disassembler even handles jump marks and offset representation. The guard enclosure features a key switch and a push button. If the button is held down while the key is turned the last downloaded code is transferred to the target PLC. If the key is turned without pressing the button the code is discarded.

C. Code Review

We store and process the disassembled code on the guard. The implementation of the review GUI is an interactive Web application, which we implemented using HTML, CSS and JavaScript, without additional server-side logic.

VII. Evaluation

In this section, we present the results of our evaluation of the guard. We evaluated two aspects of our guard. First, we evaluated its network performance, that is, whether the guard adds noticeable overhead and how much. Second, we evaluated if and how well our review process would detect attacks in different scenarios, including Stuxnet.

A. Attack Case Studies

In order to evaluate the effectiveness of the PLC Guard we applied the review checks to several example attacks and noted which checks would have revealed the malcode. This does not take maintenance changes into account but it does give a sense of the amount and types of changes that would be indicative of an attack. In what follows, we give details on the nature of the examples and their effects. Our first two examples comprise code that has been used to evaluate the academic malware targeting tool SABOT [19]. The third example is the code of the now infamous Stuxnet worm. The fourth, fifth and sixth example are attacks on our candy packaging plant that we devised and implemented for illustration and demonstration. We give a general overview of the complexity of the example programs and the example attacks in Table I.

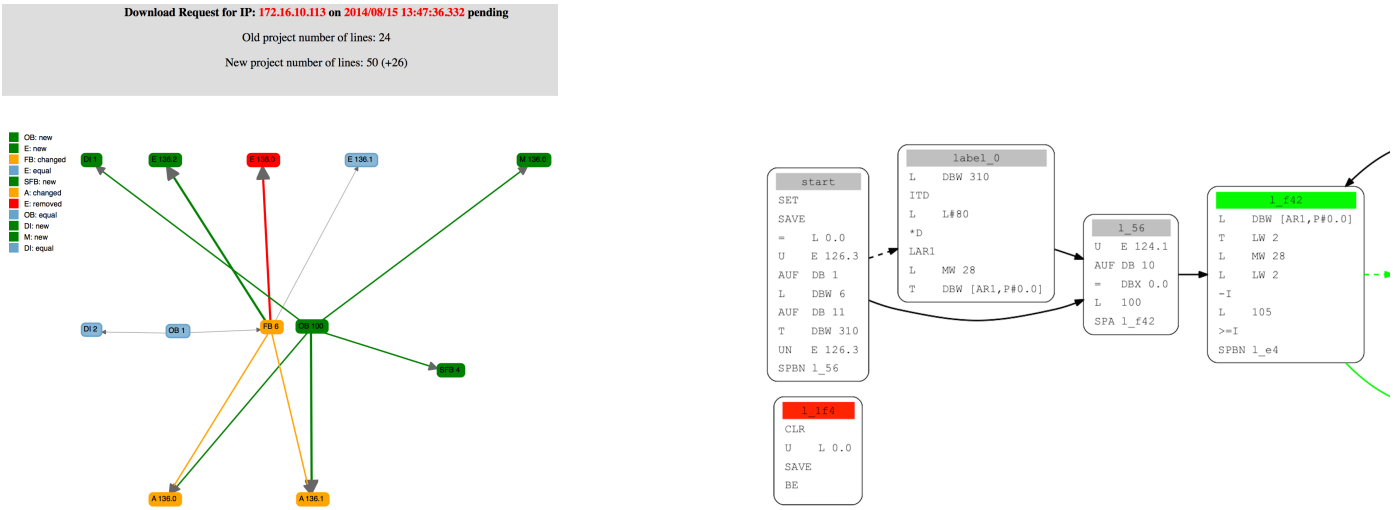

Fig. 2: Left: Inter-block comparison of two PLC programs in block level view. Right: Excerpt of an intra-block comparison. Solid lines represent the flow if the jump is executed. Dashed lines represent the alternate control flow. The block "l_1f4" was deleted and is therefore marked red. New blocks or execution paths are green.

TABLE I: Instruction count for our packaging plant, Stuxnet and the "emergency" attack from SABOT [19].

	Packaging Plant	Belt Attack	Packaging Attack	Stuxnet	Railway	Railway Attack	Traffic Light	Traffic Attack
Lines of code	3823	3844	3813	19,793	68	72	129	141
Line differences		+21 (+24, -3)	-10 (+5, -15)			8 (+6, -2)		12 (+12)
Objectblocks (OB)	2	2	2	-	1	1	1	1
Functions (FC)	0	0	0	(+43)	0	0	0	0
Functionblocks (FB)	12	12	12	-	0	0	0	0
Datablocks (DB)	33	34	33	-	0	0	0	0
Instruction								
AND (A)	232	234	230	391	12	12	22	22
NAND (AN)	30	30	30	59	12	16	6	18
OR (O)	24	24	24	77	24	24	20	20
NOR (ON)	4	4	4	0	0	0	1	1
XOR (X)	244	244	244	378	0	0	0	0
XNOR (XN)	0	0	0	0	0	0	0	0
Cond. call	0	0	0	0	0	0	0	0
Uncond. call	32	34	32	314	0	0	0	0
Cond. jump	317	318	317	917	0	0	0	0
Uncond. jump	183	183	183	1017	0	0	0	0

1) SABOT: The auhtors of SABOT [19] kindly shared with us the PLC code they used to evaluate their malware targeting tool. We selected two of the programs with the highest complexity and the "emergency" attacks [19]. The two examples consist of only one OB with no references to other blocks, for example, FC, FB or DB. By comparison with our other examples the code is fairly small and the changes that are due to the attack are prominent (see Table I). This leads us to conclude that it would be difficult to sneak the attack past a dilligent engineer who has just updated the code.

2) Stuxnet: We have access to Stuxnet code but we do not have the code that was used to drive the motor controls of the Iranian centrifuges that Stuxnet targeted. For this reason, we cannot determine the actual differences that Stuxnet introduced compared to the genuine programming. What we can determine is the size of the Stuxnext code and this what the guard would have reported as the lines of code difference. Additionally, we can estimate the size of a meaningful maintenance change. Stuxnet supports attack sequences for two types of frequency converters [20]. Based on the differences between the two converters we estimate that about 39 changes would be necessary in four blocks in order to "port" a program from one converter to the other. Even with generous room for error it is hard to believe that the addition of about 19,000 lines of Stuxnet code would have been inconspicious. Therefore we expect that Constraint 6 would have triggered. Furthermore, Stuxnet did not use correct BLD sequences and thereby violated Constraint 2, and it manipulated the DP_RECV System Function, which violates Constraint 3 (a clear giveaway).

3) Belt Attack: This attack is based on error modes we observed during the operation of our candy packaging plant By manipulating the conveyer belt we introduce problems in subsequent processing of the candy. The attack does not add or modify a significant amount of code. However, it changes nine connections to DB blocks and introduces a new function, BELT_SLOW. This is enough to trigger Constraints 5 and 8.

4) Packaging Plant Attack: In this attack, the candy packaging plant is manipulated into putting three red candies into each box. The attack overwrites the number of candies that

the user selects with a hard-coded preset. This is obviously a very simple attack. Nevertheless, it would have been prevented by Constraints 5 and 8 because it removed nine connections to DB blocks.

VIII. Related Work

Researchers have investigated a variety of strategies meant to secure cyber-physical systems, for example, new security architectures. Mohan et al. [21] presented an approach that involves detecting malcode on PLCs by means of monitoring timing side-channels. This requires exact timing profiles for the controlled systems and additional trusted hardware. While we require a trusted device as well our approach can be deployed without touching a PLC. In general, novel architectures typically have a long deployment phase. Our goal was to offer a practical approach that can be deployed quickly and easily. Cheung et al. [22] have proposed an intrustion-detection approach for SCADA systems that builds on models that characterize the expected or acceptable behavior of a system. These models are subsequently used to detect deviations from the expected behaviors due to attacks. More recently, Goldenberg et al. [23] have proposed an approach that models Modbus/TCP in order to detect intrusions in SCADA systems. Intrustion detection has a long-standing research history and there is an extensive body of literature. While intrusion detection is a reactive approach, our guard is meant to prevent intrusions. Closest to our work is the ACCAT Guard [3], [4], which we already introduced in our introduction. Guard concepts have been applied for various purposes, for example, electronic mail [24]. Our guard is probably the first instatiation of this concept for PLC code transfers between an engineering workstation and PLCs. McLaughlin [25] presented access controls for control devices whith policies for physical device behavior, which is a last line of defense. The goal of our PLC Guard is to draw the line earlier in the path of the malcode to the PLC. The Trusted Safety Verifier [2] aims to verify whether PLC code meets safety properties before allowing its transfer to a PLC. However, its lack of scalability limits its application to comparatively simple PLC code. In the following we describe our lab setup to underpin this proposition.

A. Packaging Plant

In this section, we briefly describe the miniature packaging plant we built from industrial components and from parts that we designed and produced with the help of a 3D printer. The objective of the plant is to sort and fill chocolate candy into a round metal box with a snap lid, to close the lid, and to move the box to a drop area. When a hand is placed under the box, the box drops into the hand. The box can be opened by pressing onto the lid. The color and the number of candies is configurable through an HMI. The control unit consists of a Simatic S7-313C PLC, a KTP 400 touch-sensitive color HMI, a CP 343 lean Ethernet module, a CP 341 RS-485 communication module and a PS 307 5A 24V DC power supply mounted upright on a top-at rail.

1) Process Organization: Figure 3 shows pictures of the entire setup and a picture with details of the machinery. A pneumatic cylinder with a magnet on its end extends and pulls a box from the violet stockpile. The belt moves forward until the box is under the smaller conveyer belt. A sensor registers the positions of candy on the belt. The belt signals its motor steps to the PLC, which uses this input to track the positions of candy as the belt moves them along. If candies need to be cleared off the belt, for example, because they have the wrong color or the fill level of the box has been reached then a valve opens when they are in between a nozzle and the recirculation pipe. Candy that reaches the end of the belt drops into the box. When the right amount of candy is in the box, the larger belt moves the box forward to an electro-pneumatic arm with a suction cup on its end. The box moves forward again in between two pneumatic cylinders, which shoot forward and snap the lid shut by exerting pressure on its sides. The box moves forward again until it reaches the end of the belt. A small robotic arm with two round prongs grabs the box and moves to a holding position. When a hand is extended under the box, the arm releases the box so that it drops into the hand.

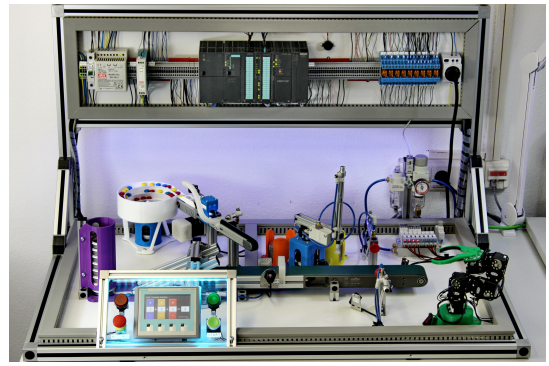

Fig. 3: Shows our miniature candy packaging plant.

2) Discussion: Our packaging plant incorporates a variety of control tasks similar to those found in industry. The focus is clearly on moving objects from place A to Z where A to Z are stages of a production process. The sensors and actuators use up most of the PLC's I/O ports and hence we believe that the plant serves reasonably well as a model of a "fully loaded" PLC. Instead of sensor-based control, processes may also be based on models that predict how conditions in a process change over time as settings change. This is useful in environments where sensing is difficult, for example, because sensors are too slow, too unreliable or exposed to harsh conditions that would negatively affect the sensor. Some sensors provide accurate readings only at a low rate. If queried too frequently, they may provide a previous reading instead, or even spurious readings. As a consequence, control programs may filter out and ignore outliers in sensor readings that would otherwise indicate high-risk conditions. This makes it difficult if not impossible to specify a safety property that is sufficiently permissive and yet prevents settings that will eventually have a detrimental effect on the process.

A second observation is that automation processes can exhibit an excessive amount of failure modes. Some failures manifest as a consequence of a lack of synchronization between different stages in a process. A comprehensive safety property must account for these interdependecies. Hence, it is not always

feasible to verify stages individually. Lastly, verifying even small control tasks quickly becomes intractable with techniques such as model checking. For example, McLaughlin et al. [2] mention that verifying their traffic light example takes 10 seconds on a desktop computer using an execution bound of 10, and takes 120 seconds at bound 14. If we fit an exponential model ($y = \alpha \cdot e^{\beta x}$) to these two data points and extrapolate then a bound of 42 will take more than 136 years to compute. Three other examples that included mentioning of a conveyer belt all had a larger complexity than the traffic light example and were evaluated at bound 6. For comparison, the motor of the smaller conveyer belt we use sends step ticks so that process control can track the positions of objects on it. One revolution of the belt (it is about 26 inches long) measures about 200 ticks, and our process tracks the positions of multiple candies on the belt. A lower execution bound for this stage of our process would be 100, the distance at which candy falls off the end of the belt. In other words, our bound cannot simply be "set higher if required for the legitimate plant functionality" [2].

IX. Conclusions and Future Work

Protecting cyber-physical systems is a challenging task. Many of the systems that exist today are vulnerable to advanced persistent threats and historical experience has taught us that trying to retrofit security to existing systems is not effective. While interesting and sophisticated ideas are investigated by researchers, it is necessary to develop practical defense mechanisms that can be deployed quickly and easily in order to prevent attacks on, for example, critical infrastructures. In this spirit, we proposed adapting a classical approach to the problem, which is a guard concept for PLC code transfers. The PLC guard is a trusted device that intercepts PLC code transfers from an engineering workstation to a PLC and delegates the decision whether or not to approve the transfer to the engineer who wrote the code. In this fashion, the guard removes control from the engineering workstation, which may have been subverted by an attacker. The guards allows engineers to compare new code with previous versions and provides various levels of graphical abstraction and summarization to ease the task of finding malicious modifications. Only the last step in the procedure would require looking at actual code differences. An analysis of six example attacks, including Stuxnet code, indicates that the summarization is effective and provides clues to the presence of malcode that can be perceived easily and efficiently. In order to arrive at realistic examples and scenarios, we implemented a miniature packaging plant. We expect that the packaging plant project will be a useful tool for further exploration of attacks and defenses on cyber-physical systems.

References

[1] R. Langner, "To Kill a Centrifuge- A Technical Analysis of What Stuxnet's Creators Tried to Achieve," The Langner Group, Tech. Rep., 2013.

[2] P. M. Stephen McLaughlin, Devin Pohly and S. Zonouz, "A trusted safety verifier for process controller code," in *Proc. NDSS*, 2014.

[3] S. R. Ames and D. R. Oestreicher, "Design of a message processing system for a multilevel secure environment," in *Proc. National Computer Conference*, vol. 47, AFIPS. AFIPS Press, 1978, pp. 765–771.

[4] J. P. L. Woodward, "Applications for multilevel secure operating systems," in *Proc. National Computer Conference*, vol. 48, AFIPS. AFIPS Press, 1979, pp. 319–328.

[5] J. P. Anderson, "Computer security technology planning study. volume 2," DTIC Document, Tech. Rep., 1972.

[6] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[7] L. Fraim, "Scomp: A solution to the multilevel security problem," *Computer*, vol. 16, no. 7, pp. 26–34, July 1983.

[8] C. Weissman, "Blacker: security for the ddn examples of a1 security engineering trades," in *Research in Security and Privacy, 1992. Proceedings., 1992 IEEE Computer Society Symposium on*, May 1992, pp. 286–292.

[9] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler, "A system for graph-based visualization of the evolution of software," in *Proceedings of the 2003 ACM Symposium on Software Visualization*, ser. SoftVis '03. New York, NY, USA: ACM, 2003.

[10] Q. Tu and M. W. Godfrey, "An integrated approach for studying architectural evolution," in *Proceedings of the 10th International Workshop on Program Comprehension*, ser. IWPC '02. Washington, DC, USA: IEEE Computer Society, 2002.

[11] M. D'Ambros and M. Lanza, "Software bugs and evolution: a visual approach to uncover their relationship," pp. 10 pp.–238, March 2006.

[12] T. Khan, H. Barthel, A. Ebert, and P. Liggesmeyer, "Visualization and evolution of software architectures," in *OASIcs-OpenAccess Series in Informatics*, vol. 27. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[13] K. B. McKeithen, J. S. Reitman, H. H. Rueter, and S. C. Hirtle, "Knowledge organization and skill differences in computer programmers," *Cognitive Psychology*, vol. 13, no. 3, pp. 307–325, Jul. 1981.

[14] A. F. Norcio, "Indentation, documentation and programmer comprehension," in *Proc. CHI*, 1982, pp. 118–120. [Online]. Available: http://doi.acm.org/10.1145/800049.801766

[15] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "Program indentation and comprehensibility," *Commun. ACM*, vol. 26, no. 11, pp. 861–867, Nov. 1983. [Online]. Available: http://doi.acm.org/10.1145/182.358437

[16] G. K. Rambally, "The influence of color on program readability and comprehensibility," *SIGCSE Bull.*, vol. 18, no. 1, pp. 173–181, Feb. 1986. [Online]. Available: http://doi.acm.org/10.1145/953055.5702

[17] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proc. ISSTA*, 2008, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390647

[18] ——, "Automatically documenting program changes," in *Proc. IEEE/ACM ASE*, 2010, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859005

[19] S. E. McLaughlin and P. McDaniel, "SABOT: specification-based payload generation for programmable logic controllers," in *Proc. ACM CCS*, 2012, pp. 439–449.

[20] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet Dossier - Version 1.4," Symantec Corporation, Tech. Rep., 2011.

[21] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, "S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems," in *Proc. ACM International Conference on High Confidence Networked Systems*, ser. HiCoNS '13. ACM, 2013, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/2461446.2461456

[22] S. Cheung, B. Dutertre, M. Fong, U. Lindqvist, K. Skinner, and A. Valdes, "Using model-based intrusion detection for scada networks," in *Proc. SCADA Security Scientific Symposium*, 2007.

[23] N. Goldenberg and A. Wool, "Accurate modeling of Modbus/TCP for intrusion detection in SCADA systems," *International Journal of Critical Infrastructure Protection*, vol. 6, pp. 63–75, 2013.

[24] S. D. Wolthusen, "A distributed multipurpose mail guard," in *Proc. IEEE Workshop on Information Assurance and Security*, United States Military Academy, West Point, NY, Jun. 2003.

[25] S. McLaughlin, "Cps: Stateful policy enforcement for control system device usage," in *Proc. ACSAC*. ACM, 2013, pp. 109–118. [Online]. Available: http://doi.acm.org/10.1145/2523649.2523673