INVESTIGATING ATTACKS ON INDUSTRIAL CONTROL SYSTEMS USING DETERMINISTIC REPLAY SIMULATION

A Thesis

Submitted to the Faculty

of

Purdue University

by

Greg Walkup

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. Dongyan Xu, Chair

Department of Computer Science

Dr. Eugene Spafford

Department of Computer Science

Dr. Xiangyu Zhang

Department of Computer Science

Approved by:

Dr. Voicu Popescu

Head of the School Graduate Program

ACKNOWLEDGMENTS

At Purdue, Prof. Dongyan Xu helped give the project direction, gave valuable insights, and supported it from start to finish. Abdulellah Alsaheel and Cody Butler also supplied their comments and advice. Thanks also go to Profs. Zhang and Spafford for agreeing to serve on the thesis committee.

This work was supported financially by Sandia National Laboratories. At Sandia, Vincent Urias, Han Lin, and Kelcey Tietjen supported the project, with some advice also coming from Wellington Lee.

Thank you all for your help and support!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

PLC Programmable Logic Controller

RTU Remote Terminal Unit

HMI Human-Machine Interface

ABSTRACT

Walkup, Greg M.S., Purdue University, May 2019. Investigating Attacks on Industrial Control Systems Using Deterministic Replay Simulation. Major Professor: Dongyan Xu.

From factories to power grids, industrial systems are increasingly being digitally controlled and networked. While networking these systems together improves their efficiency and convenience, it also opens them up to attack by malicious actors. When these attacks occur, forensic investigators need to quickly be able to determine what was compromised and which corrective actions should be taken. In this thesis, a method is proposed for investigating attacks on industrial control systems by simulating the logged inputs of the system over time using a model constructed from the control programs that make up the system. When evaluated, this led to the detection of attacks which perturbed the normal operation of the system by comparing the simulated output to the actual output. It also allowed for dependency tracing between the inputs and outputs of the system, so that attacks could be traced from their unwanted effects to their source and vice-versa. This method can thus greatly aid investigators in recovering the complete attack story using only logs of inputs and outputs to an industrial control system.

1 INTRODUCTION

A control system is a system which controls some part of the physical world. As control systems become more and more complex and interconnected, the need to secure such systems has increased greatly. The connection of such systems to the wider internet has also greatly increased their attack surface. This is particularly true for industrial control systems, which are being connected to the internet to cut costs and preserve ease of use, and which often control heavy machinery used in high-value processes. [1]

Attacks such as Stuxnet [2] and Crashoverride [3] show that attackers are actively targeting industrial control systems. It is therefore important that the owners of such systems are able to detect and investigate these attacks. However, due to the wide variety of hardware vendors and system configurations, it is difficult to gather enough domain knowledge and forensic expertise in one place to quickly and accurately investigate an attack. [4]Furthermore, shutting down a control system while an incident is being investigated may be impossible or cost large amounts of money in lost revenue. [5]

In order to successfully investigate an attack on a control system, investigators must determine which part of the system failed (if any), and must be able to discover the reason for that part's failure. In some cases, it may not even be clear that an attack has occurred, or what its effects were. [6] If it is determined that an attack did occur, investigators need to find out the root cause of the failure within the system and trace that cause back to a physical input or traditional IT system to continue the investigation through more traditional methods.

In this work, we propose a method of investigating attacks on industrial control systems by automatically creating a model of the system by analyzing the code of the controllers in the system. By logging all data exchanges between the system and the

outside world, a description of the system's historical interactions with the physical process and its operators can also be constructed. The system can then be simulated using these logged values in order to provide a baseline for correct system behavior. This simulation can also be used to trace the causality of data as it flows through the system. This investigation framework thus seeks to provide investigators with knowledge of (1) what happened in the system (2) what should have happened in the system and (3) why something happened in the system.

2 LITERATURE REVIEW

Much work has been done in the area of attack detection and monitoring in industrial control systems. Most approaches to intrusion detection and prevention either use a learning-based model to determine the correct behavior of the system, or use a specification provided by domain experts. Using one of these model types, attacker actions can be differentiated from benign actions, triggering a spectrum of warnings and remedial actions. While much of the work does not have to do with forensic investigation, detecting an attack in real time and investigating an attack that happened in the past can use similar techniques for discovering attacker actions.

Many learning-based approaches([7] [8] [9]) perform anomaly detection solely on the data in a control system. This minimizes the performance overhead involved in monitoring for anomalies and reduces or eliminates the need for a formal specification. This ease of use is balanced by their inability to take advantage of knowledge of how the larger system operates. Hadziosmanovic et al. [10] created an IDS that extracts semantic information from networked control protocols and attempts to forecast the behavior of each variable in the system. This allows for detection of attacker actions as anomalies which do not match the forecasted model. Other works expand on the idea of anomaly detection by selective probing of other variables not passed on the network [11], looking for suspicious sequences of events [8], and by decoding even unknown control system protocols [12]. Barbosa et al. [13] also propose a method for whitelisting flows of data rather than sniffing packets for individual variables.

Other approaches focus on creating a specification for how the system is supposed to operate using expert knowledge. This makes monitoring more configurable and allows tailored detection based on user-specified alarms. However, creating such a specification is time-consuming, especially for complicated systems, and it can be difficult to generalize approaches across different processes and hardware vendors.

Wang et al. [14] describe a system for detecting false data injection attacks. The users provide a specification of the system, either directly or through analysis of control firmware. This specification is used to create a state transition diagram which describes the valid behavior of the system. Deviations from this state transition model can then be detected and flagged for anomalous behavior. Some approaches [15] propose using redundant controllers with identical programming in place of a formal model of the system. Fauri et al. [16] propose a method that uses a hybrid approach, with expert intervention being used to derive and refine features for anomaly detection. Another approach by Hadziosmanovic et al. [17] constructs a formal model of the system with methods borrowed from safety research and combines it with pattern matching on log entries to attempt to detect suspicious behavior.

Some efforts also focus on using the given model of the system to pre-emptively stop attacks from reaching the physical process. Lerner et al. [18] give an approach for vetting configuration changes to a controller before they are implemented by forecasting future states of the system based on the changes. A similar effort by McLaughlin et al. [19] vets PLC programs before they are uploaded to a controller to see if it is possible for them to violate the system's safety rules. Chiluvuri et al. [20] describe a system where the trusted verification hardware is instead between the physical process itself and the controller, preventing unsafe behavior even if the controller is compromised.

Work has also been done in incorporating the physics of the process being controlled into a model. This allows for more precise detection of anomalies, but also requires more analysis by subject-matter experts. Physical models can also be difficult to generalize for use in more than a single process. Giraldo et al. [21] provide a survey of methods that use physical models. Many of these models rely only on an abstracted model of the system, which removes some of the differences between specific implementations. Ghaeini et al. [22] and Do et al. [9] both use Cumulative Sum (CUSUM) to detect anomalies in the physical process itself. Mo et al. [23] and Krotofil et al. [24] describe systems for specifically detecting falsified sensor data.

Crdenas et al. [25] perform a case study on the simplified Tennessee-Eastman process, a theoretical industrial process that involves a single chemical reaction. They propose several types of attacks, and show the resiliency of the process to stealthier attacks. They also show demonstrate how a mathematical model based on the physical process itself can be used to detect attacks that significantly perturb the system. Urbina et al. [26] provide a survey of various attack detection techniques based on the physics of the system being controlled. They also propose an attacker model which can always remain undetected, given perfect knowledge of the detection method used. In this case, the limiting factor is instead how much damage the attacker can inflict for a given detection method while remaining under the threshold.

3 METHODS

3.1 Abstractions and Modeling

Control System For our purposes, it is convenient to abstract an industrial control system as a collection of variables. These variables can represent sensor readings from the physical world, user input, configuration settings, or the state of the control output of the system. This allows us to strip away a significant amount of vendor- and system-specific configuration and generalize a model across a broad spectrum of control systems. This is an approach also taken by previous work ([14] [10] [25]).

Controller Each variable in an industrial control system is considered to reside in a *controller*, which is responsible for manipulating it or using it to make decisions.

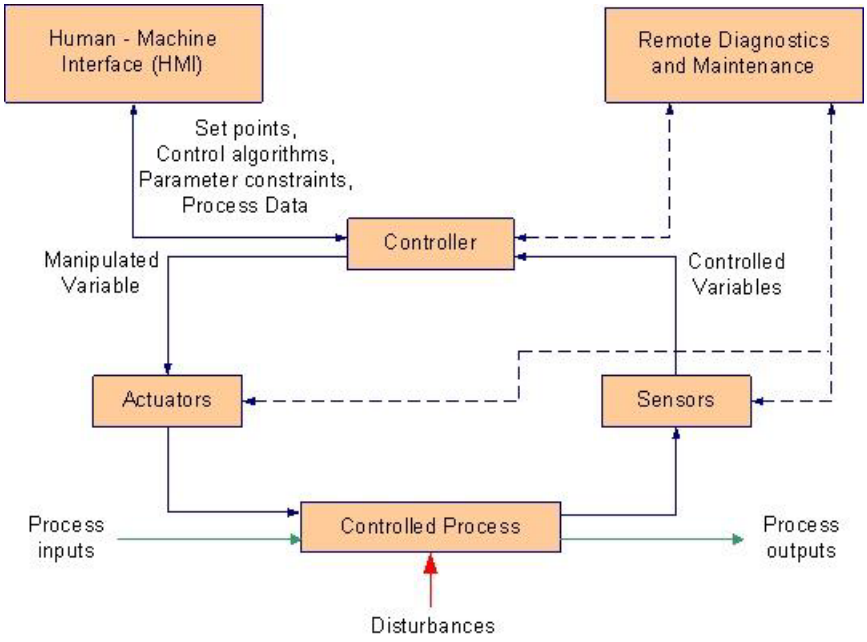

Figure 3.1. A diagram of a simple industrial control system from [6]

In the real world, controllers correspond to PLCs (Programmable Logic Controllers), RTUs(Remote Terminal Units), or any other low-level device which is capable of aggregating or manipulating data. Variables are not implicitly shared between controllers, but two controllers can have a variable that corresponds to the same logical value.

Control Program A *control program* is an sequence of variable operations attached to a controller. A control program can manipulate variables within the controller in which it resides, and can be run continuously, on a timed loop, or in response to a particular variable state. The control program(s) in a controller represent the automation logic present within that controller's real-world analogue. Programs are assumed to be deterministic with regard to their input variables.

Log Entries Each action taken in relation to a controller can be abstracted to an operation on a variable. For example, a controller may read in the value of a sensor to one of its variables and write out the value of a control valve from another variable. Values can be "written" to a controller to change the value of one of its variables or out of a controller to move the value of one of its variables to outside the system or another controller. Values can also be "read" from the controller to get the value of one of its variables or "read" to the controller to update the value of one of its variables. These "read" and "write" closely mirror the format of industrial protocols and can fully represent data moving through the system in our abstracted model. This approach is shared by some previous work ([11] [26]).

To unambiguously identify each operation, in addition to metadata we must log (i) the source device (ii) the destination device (iii) the address of the variable(s) being manipulated (iv) the data being transferred. If the data is being transferred over a network of some kind (as is common in modern industrial control systems), the source and destination device can be identified by their network addresses. The address of the targeted variable and the actual data being transferred depend on the specific protocol being logged. An example of the log format is provided in figure 3.2.

Figure 3.2. Example log format

Input from the physical world (e.g. from sensors) or input from an operator-controlled HMI (Human-Machine Interface) are considered to be "outside" the system. The values of these inputs are considered to be part of the system only once one of these operations has moved the values into a controller variable. This abstracts away differences between sources of input and mirrors the real-world behavior of the controllers, which operate on data locally before sending out output values.

3.2 Attacker Model

We model an attacker that seeks to influence the high-level execution of the process in some way. We assume that an attacker may compromise the execution logic of one or more controllers and may also be able to manipulate data as it flows through the system. While an attacker may be able to avoid having their actions directly logged, we assume that an attacker is not able to arbitrarily manipulate or delete log entries. We also assume that all actions performed *by* a controller (though not necessarily all actions *on* a controller) are correctly logged, so that each controller's real behavior is reflected in the collected logs. Finally, we assume that an attacker's influence on the system starts after the start of logging, and thus that their actions or the secondary effects of their actions are logged.

The other major assumption that we make is that investigators are able to procure a "clean" copy of the controller program source code, or otherwise acquire a model input that accurately reflects the "correct" behavior of the system. Since the reference

controller code is used to efficiently construct a model of normal system behavior, this model will be incorrect if an attacker has managed to significantly corrupt it to redefine the system's behavior. We feel that this is a reasonable assumption, given that a model for the system behavior must originally come from somewhere.

3.3 Analysis

3.3.1 Log Collection

In a modern industrial control system architecture, the system often interacts with sensors, actuators, and operators over a network of some kind. Logging each operation can then be performed simply by sniffing the network traffic and extracting protocol-specific fields [10]. This allows for log collection that is unobtrusive and does not significantly impact the performance of the system. If certain variables are not exchanged over the network, or are modified in some other fashion, selective probing of those variables can also fill in the gaps to provide a more complete historical view of system behavior [11].

As discussed in section 3.1, logged actions at the control system level correspond to "read" and "write" operations on variables in the system. Since various industrial protocols encode this information in different ways, it is necessary to decode each protocol before converting relevant messages to a uniform log format (i.e. one that records read and write operations). After collection, these log entries can be combined together to create a single unified log for the whole system.

Given these logs, we now have access to the raw data associated with the past performance of a given control system. However, these logs by themselves do not provide the necessary details for a forensic investigation. They do not provide a picture of the internal state of each controller, only the inputs and outputs produced by them. While the logs may provide a picture of what occurred, they shed no light on why something occurred or whether something was the result of normal system behavior or something more malicious.

3.3.2 Model Construction

A straightforward way to mitigate these problems is to stop treating the controllers in the system as black boxes. If we can simulate expected controller behavior, we can determine a baseline for system behavior that can be compared to the behavior of the real historical system. This process can also aid in "connecting the dots" by showing which inputs of the system affect which outputs. In this way, the internal operations of the system's controllers can be inferred without directly logging them, which could be prohibitively expensive from a performance standpoint [27].

As previously mentioned, each physical controller is broken down into a set of variables and a set of control programs. The control programs themselves are gathered directly from the code executed on the given physical device (e.g. a PLC) and ingested to form an execution model. This model is composed of a hierarchical set of function calls, with the leaf nodes in the hierarchy being basic operations provided by the controller (e.g. add, subtract, move). The models of each program may read and manipulate the variables that are part of the controller, and may also contain internal variables accessible only inside the controller. All controllers also share a read-only time variable that represents the current simulation time.

PLC programs are often written using graphical languages like function block diagrams or ladder logic, both defined in IEC 61131-3 [30] (the IEC standard for programming languages in PLCs). An example from our test system is shown in figure 3.3. For our evaluation system in section 4, the written form of programs comes from the PLCopen TC6 XML format [31] designed for representing programs following the programming language standards in IEC 61131-3. This format is, however, far from universal, and so a manual effort would have to be made to support other forms of controller programs.

Control programs usually consist of program logic that is executed in a loop. For our purposes, they will be normally simulated as executing on a timer; a specific program will execute its logic periodically on a set interval, either taken from the

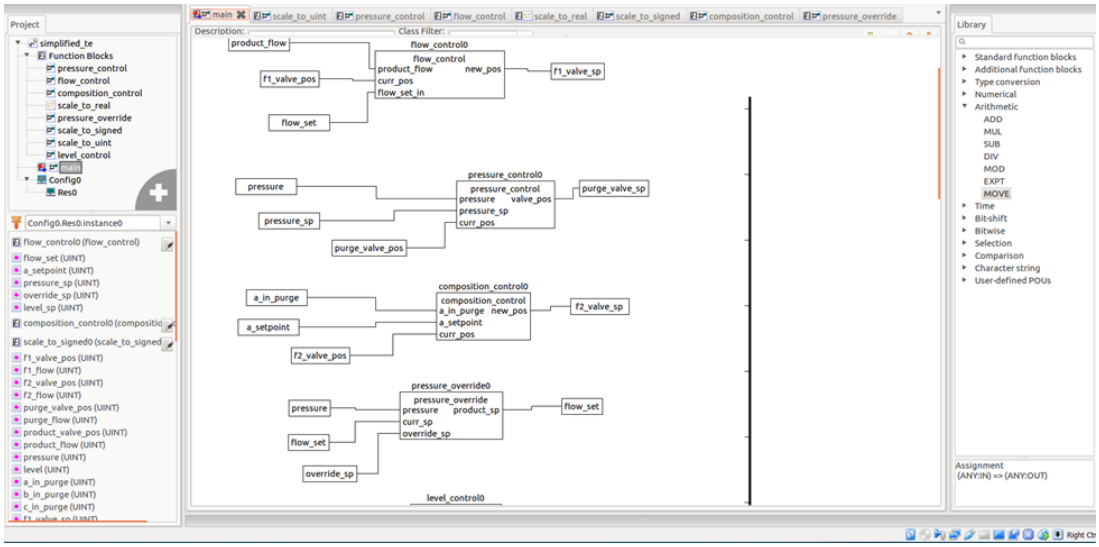

Figure 3.3. The PLC program from GRFICS [28], opened in the PLCOpen editor [29]

program specification or specified by the user. It is possible for a control program to run at other times (e.g. activating when a certain condition occurs), but this is not implemented in the current iteration of our model.

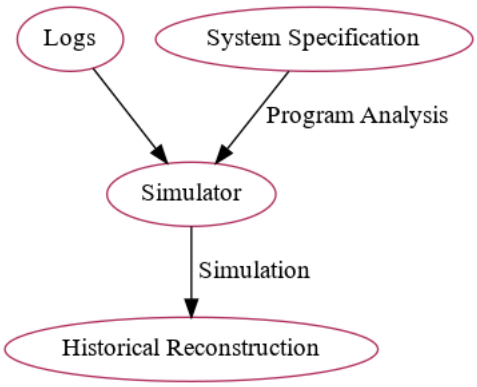

Figure 3.4. Simulator workflow

3.3.3 Simulation

After the logs are ingested and the system model constructed, the entire system is simulated in a discrete event simulator. Events in the simulator consist of log events, which occur at the time given in their log entry, and program execution events, which occur periodically as they are specified. While a program execution may not occur quickly enough to be a discrete event, in many systems the actual variables are exchanged over the network all at once at the end of an execution cycle instead of continuously. In either case, simulating executions of a program as discrete events serves well enough for our purposes.

When a log event describing data flowing into a controller is simulated, the corresponding variable in the execution model is simply updated, ready to be used when that controller's program(s) next execute. When a log event describing data flowing out of a controller is simulated, the value itself is checked against the value in the simulation. If the simulated value and the actual value from the log differ by more

than a certain threshold, an error is generated and logged. This error threshold is configurable and is based on the total range of the variable during the observed period. Warning thresholds can also be set to show less severe deviations. If the logged value is within the threshold, the variable is noted as having successfully been verified.

When a program execution event is simulated, the logic of the program corresponding to the event is executed. The program is executed by performing a depth-first traversal of the execution hierarchy, reading and updating the controller's variables along the way. At the bottom of the execution hierarchy are leaf functions that represent the basic functions of the controller being simulated (e.g. add, subtract, compare). These functions are manually specified based on the manufacturer's specifications rather than being read in from the controller's program. If the program executes on a timer, a new execution event is also generated in the future after the appropriate interval.

In addition, whenever a log event changes a variable, a unique id corresponding to that log event is added to the trace set of the variable in question. Whenever that variable's value is used to compute another value during the execution of a controller program, the trace set follows the value to the new variable. This trace propagates forward to all variables that ultimately end up being affected by the value written in that input event. For example, if one variable is assigned to another, the first variable's trace set would be copied to the second. If two variables are added together with the result being stored in a third variable, the third variable's trace set would be the union of the first two variables' trace sets. Whenever an output event occurs, this trace set can then be extracted and saved, connecting the output event to the input events that influenced its value. This allows for the dynamic tracing of data through the system as it executes.

This simulation relies on the fact that control systems are usually built as deterministically as possible in order to maximize the reliability of the controlled process. This makes deterministic simulation of the system's behavior practical [32] when the inputs and outputs of the system are logged in their entirety. This high degree of de-

terminism makes this simulation approach more feasible in industrial control systems than it would be in traditional computing systems.

3.4 Investigation

In a forensic investigation, investigators will generally examine log data in order to determine what happened after some triggering event involving suspicious behavior on the part of the system or a suspected attack originating from outside the system. An investigator may also examine logs to determine if an attack actually occurred during the time period given. We now split attacks into two categories, which are investigated differently: **logic-based attacks** and **data-based attacks**.

3.4.1 Logic-Based Attacks

A logic-based attack is an attack that tampers with the control logic of one or more controllers. These controllers then behave in a manner that is different from the "reference" program provided to the simulator. As a result, if an altered control program produces a different output than the reference program would have, the simulator will take note and generate a notification that the expected output and the actual output do not match.

If the output is close, but not identical, to the expected output, the simulator will classify it as normal behavior, a warning, or an error based on configurable thresholds. More strict thresholds will result in more false positive errors from noise, while more lax thresholds may result in missing the influence of an attacker's alterations. Note that if the altered control program always produces the same expected outputs as the original control program in the logged time period, the attack does not actually affect the modeled system.

After it finishes running, the simulator produces a timeline showing the output variables in the system and the points at which they were in a "expected" state, a "warning" state, or an "error" state. These thresholds are configurable; by default a

"warning" is recorded if the difference is more than 1% of the recorded range of the variable and an "error" is recorded if the difference is more than 10% of the recorded range of the variable. See section 4 for examples of such timelines.

3.4.2 Data-Based Attacks

A data-based attack is an attack that tampers with the data flowing into controllers. In these attacks, the behavior of the system is unaltered when compared with normal system behavior. Instead, an attacker modifies the input data (e.g. from sensors) or issues commands that could have been issued by an authorized user. In both of these cases, the simulator will generally show the system operating normally, since the control logic of the controllers has not changed.

In order to investigate these types of attacks, an investigator will need some starting point in the system which he suspects has been altered by an attacker. This can come in the form of a suspect IP address, sensor reading, or user command. If the investigator provides this starting point to the simulator, the simulator will first identify which input log events correspond to the given criteria. The simulator can then trace the influence of those particular log events through the program, which reveals the outputs which were affected by the suspected malicious input. These malicious input points may be discovered by more traditional enterprise forensics, or through the use of previous work which focuses on identifying anomalous sensor inputs ([26] [9] [23]).

Similarly, if the investigator identifies one or more suspicious outputs (e.g. a open valve which causes a tank overpressure), they can use the simulator's tracing to determine which logged inputs had an influence in producing the given output. For each output, the simulator logs which input log events influenced the value that was output, based on the results of the simulation. This will identify some set of user actions and sensor readings which directly contributed to the given output value. In this way, the simulator helps cull the list of relevant log entries that the investigator needs to

look at in order to determine the root cause of an undesirable event. Examples of both types of tracing (forward and backward) are given in section 4.5.

In some cases, an attacker may give input to a controller in such a way that the input is not logged (e.g. by inputting values manually using the controller's physical interface). In this case, the tracing would not show that attacker's influence because it is not part of the logs. However, since the simulation is being performed over log values, this means that the logs will record the real controller not behaving like the simulated controller because the real controller is operating on the attacker's input, which the simulated controller does not have. Thus, this type of attack can be investigated in a similar fashion to a logic-based attack.

4 EVALUATION

We tested our approach using GRFICS [28], a framework designed to simulate the Simplified Tennessee Eastman process. In this process, two reactants are mixed with a non-reactant gas in a tank, where they condense into a product, which is pumped out of the tank. We ran a baseline test, along with several attacks on this framework.

4.1 Experimental Setup

GRFICS consists of three virtual machines: one machine running the emulation of the physical process, one machine acting as a PLC, and one machine acting as an HMI. In our setup, a fourth machine also passively collects logging data as documented in section 3.3.1.

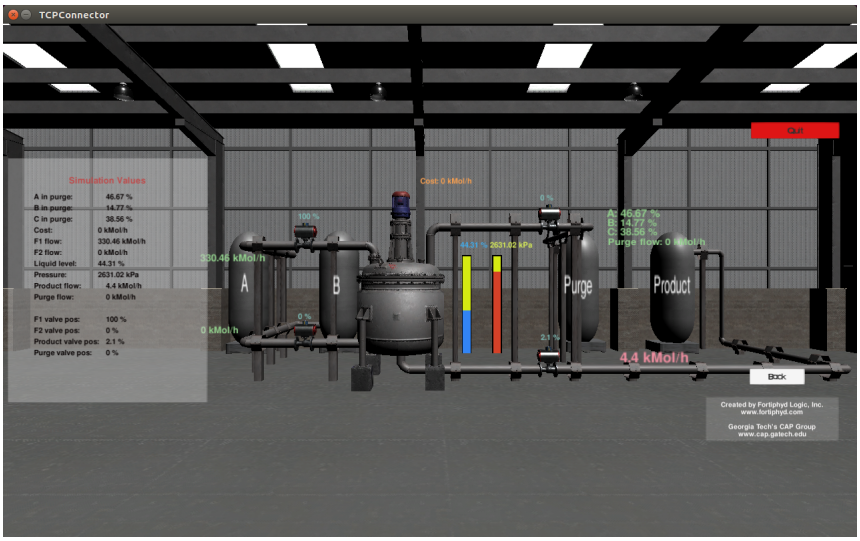

Figure 4.1. The graphical portion of the GRFICS simulator

The VM of the simulation includes a graphical component written in the Unity Engine [33] that shows the current physical status of the plant. The VM containing the PLC runs a version of OpenPLC [34] which emulates a PLC that communicates using the Modbus/TCP protocol [35]. This PLC was used to control the simulated environment on the first VM over the network. The third VM contains a custom HMI built in Advanced HMI [36], a free framework for building an HMI in .NET. The error rates for each run are shown in table 4.1.

All tests were performed by first starting the logging mechanism, which passively sniffed packets traveling over the shared LAN used by all VMs in the system. Then, the physical process simulation was started, followed by the OpenPLC program and the HMI. This ensured that no log data was missed and that the actions of the PLC were fully captured. The system was then allowed to run for a set period of time, or until the attack had completed (usually 10-15 minutes), at which point the logs were saved for analysis.

Table 4.1.

Simulator Trial Results

Trial	Correct	False Error	False Warning	Accuracy
Benign Run	73675	25	742	99.0%
Program Attack	85815	24	569	99.3%
Network Attack	147398	1988	8666	93.3%
Data Attack	78566	16	854	98.9%

4.2 Benign Test Run

In the benign test, the system was simply left to run by itself for about 10 minutes with no attack being performed. This was to test how accurately the behavior of the system could be simulated over normal conditions. Overall, 742 log entries were

flagged as warnings (more than 1% deviated, but less than 10%) and 25 log entries were flagged as errors (more than 10% deviated), leading to a total false positive rate of 1.0%. The results are shown in table 4.1 and figure 4.2.

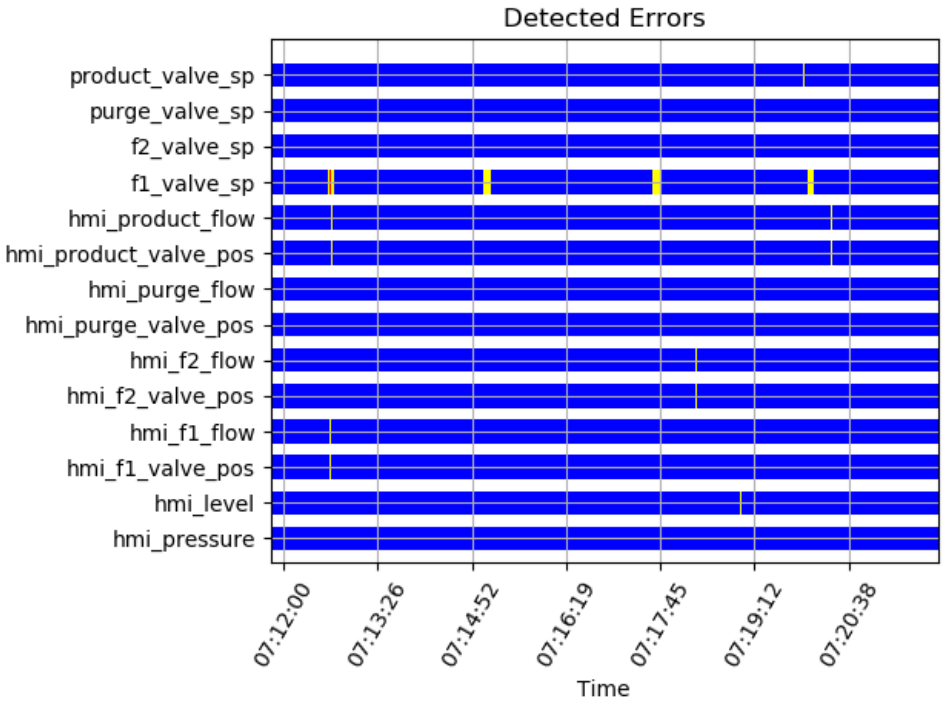

Figure 4.2. Error Timeline for Benign Run

In general, most detection errors stem from places where outputs change very quickly. This is most pronounced in the variable *f1_valve_sp*, which is a variable corresponding to the position of a valve that opens and closes very quickly. In these cases, the simulated outputs are slightly out of sync with the real outputs, with the large changes causing this slight synchronization error to trip a warning or error. Some other detection errors also stem from slight state differences when the controller starts (toward the left of the timeline). Those errors go away once the simulation and controller have had some time to converge. In general, the false warnings are transient enough to not be too suspicious, and investigators can customize the warning and

error thresholds to better detect subtle attacks or remove false positives depending on the sensitivities of the application.

4.3 Program Modification Attack

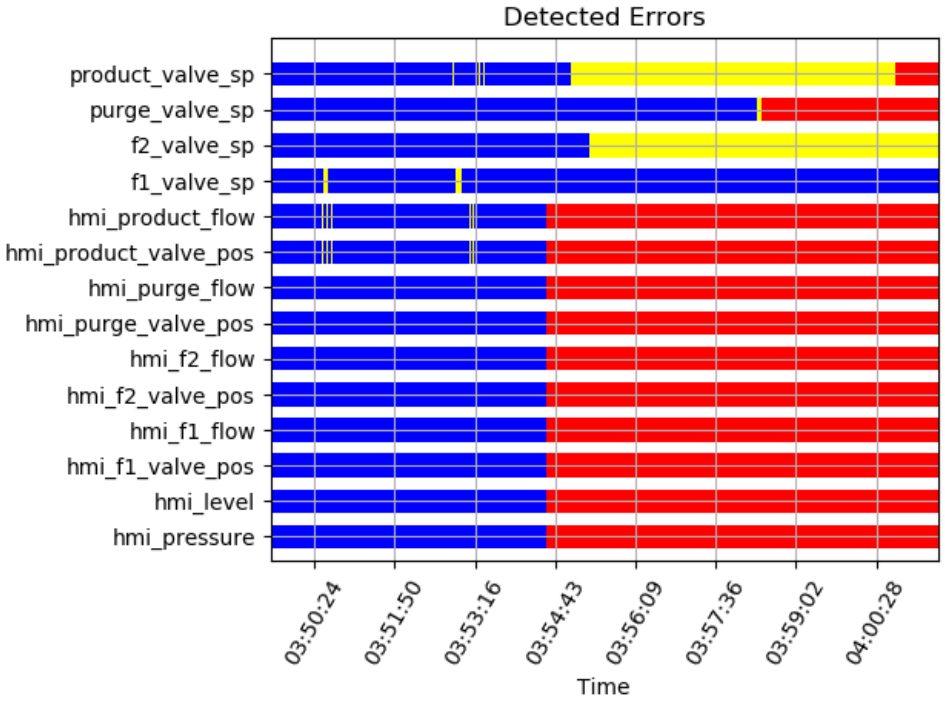

Figure 4.3. Error Timeline for Program Modification Attack

In this test, the system was run as in the benign test, except that after five minutes the program on the PLC was modified maliciously so that the tank would eventually overpressure and explode. The modified program set all of the outputs to fixed values, effectively sticking both the input valves to the tank open and both the output valves of the tank closed, causing a pressure buildup. The attack also modified the HMI outputs so that an operator looking at the HMI would not immediately notice anything wrong. The attack successfully caused the simulated tank to explode.

The effects of the attack were fairly obvious, with the simulator easily detecting the malicious values. The results are shown in table 4.1 and figure 4.3.

Note that in this case, while the fake HMI values and the product valve are quickly marked as erroneous when the attack begins, other simulated outputs take some time to diverge from the simulated values, or do not diverge at all. This is because those outputs happen to be the same (or similar) in the attacker's program and the original program, with the differences between them still being sufficient to cause the tank to explode. However, once they are detected, the values are clearly suspicious because they are a sustained deviation from normal values over a long period of time. The simulated and actual values of the purge valve setpoint, which is supposed to open to relieve pressure when it reaches unsafe levels, are shown in figure 4.4.

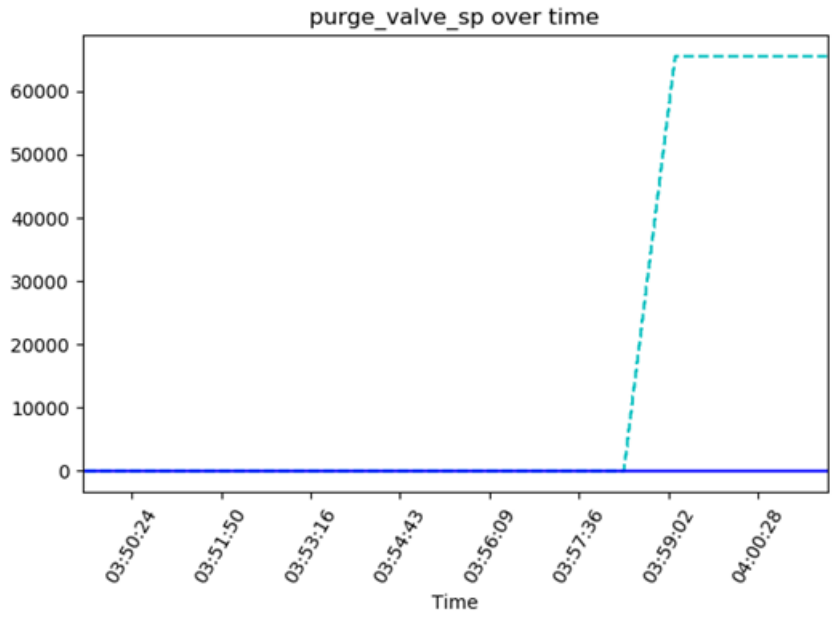

Figure 4.4. Actual logged value of the purge valve setpoint (solid line) vs. the simulated value (dashed line)

4.4 Network Attack

In this test, the system was again run normally for five minutes, at which point a fifth malicious machine on the network began sending commands directly to the valves in the physical simulation. These commands were to open the two valves leading into the tank and close the two valves leading out of the tank, with the goal being to cause a tank overpressure. While the PLC was still sending correct commands to the valves, the attacking machine sent them faster, with the result being that the tank exploded anyway. The results are shown in table 4.1 and figure 4.5.

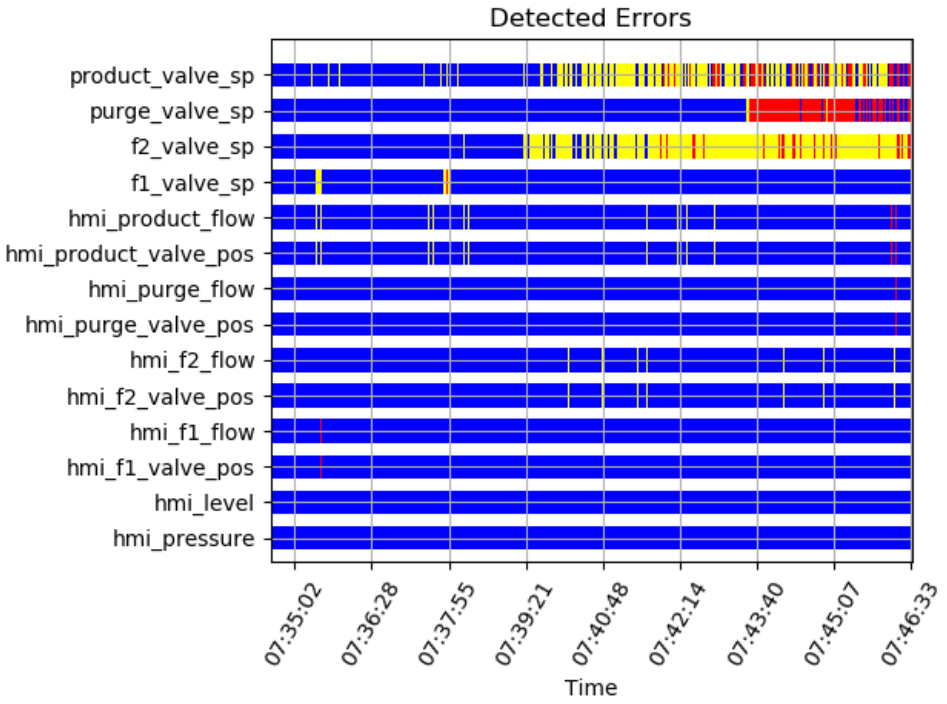

Figure 4.5. Error Timeline for Network Attack

The results of this test were a bit spottier, since the logs showed both the correct commands from the PLC and the incorrect commands from the attacker. In this case, it flagged the values from the attacker as incorrect and the values from the PLC as correct, but since they were interspersed, the appearance of the graph is a

little more varied. In addition, this attack scenario resulted in significantly more false positives (about 7% vs. about 1% in the previous two cases). Upon investigation, this was primarily because the large changes in valve position (fully open, then fully closed, etc.) were caused by the network attacker and the PLC fighting for control of the valve. This caused large changes in valve position very rapidly, which, as noted above, tends to create false positives as the simulation may be slightly out of sync with the actual run by some fraction of a second. Nevertheless, the attack was not very stealthy, since even a cursory examination of the log data points to the attacker-controlled machine as the source of the incorrect commands.

4.5 Program Data Attack

This attack was intended to test the tracing capabilities of the simulator. In this scenario, the HMI machine was compromised by replacing a communications DLL used by the HMI program with a malicious substitute. Every so often, this malicious DLL sent a command to the PLC to change the pressure setpoint of the tank. This caused the purge valve to open, wasting the raw material of the reaction unnecessarily. The compromised communications library then lied about the status of the valve to the HMI program, concealing the attack from operators. The goal of this attack was not to make the reactor explode, but rather to stealthily waste resources over a long period of time.

Unlike the previous two attacks, the simulator did not notice any incorrect behavior(fig. 4.6) because the system was still functioning correctly; in this case the attacker is using a valid command from the (compromised) HMI to accomplish their goals. In this scenario, the simulator instead aims to help an investigator either pinpoint the cause of an undesirable behavior (like the purge valve opening when it shouldn't) or determine the effect of a known malicious component of the system (such as the compromised HMI), since it might be difficult to determine whether an action is malicious if that action mimics normal operator behavior.

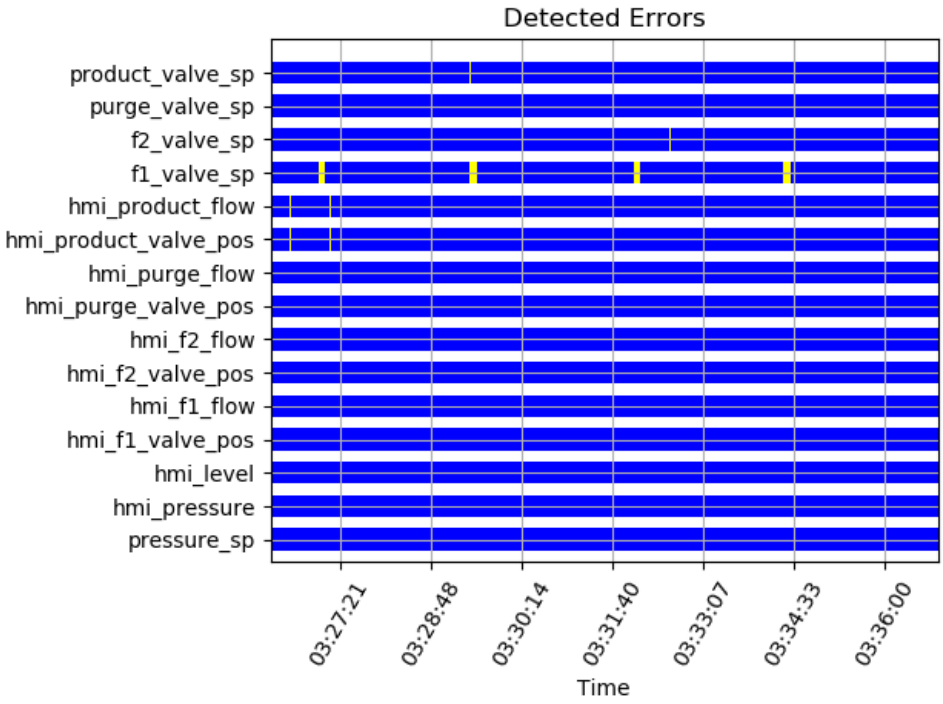

Figure 4.6. Error timeline for stealthy program data attack (virtually identical to benign case)

When tracing the effect of the compromised HMI forward, the simulator correctly identifies that the variable *pressure_sp* (the pressure setpoint of the tank) is modified by the malicious commands, and also that *purge_valve_sp* (the outgoing setpoint of the purge valve) is also affected as a second-order effect of changing the pressure setpoint. This is shown in figure 4.7. This information is obtained by marking all logged actions originating from 192.168.95.3 (the IP of the compromised HMI) and highlighting the variables that depend on those log entries.

When tracing backwards from the opening purge valve, the simulator can identify all the influences on the purge valve setpoint at the moment it opens. These influences are shown in figure 4.8. While many of these (the ones that start with underscores) are values that are part of the program itself, three represent log entry IDs. These three log entries correspond to the sensor reading of the previous position of the purge

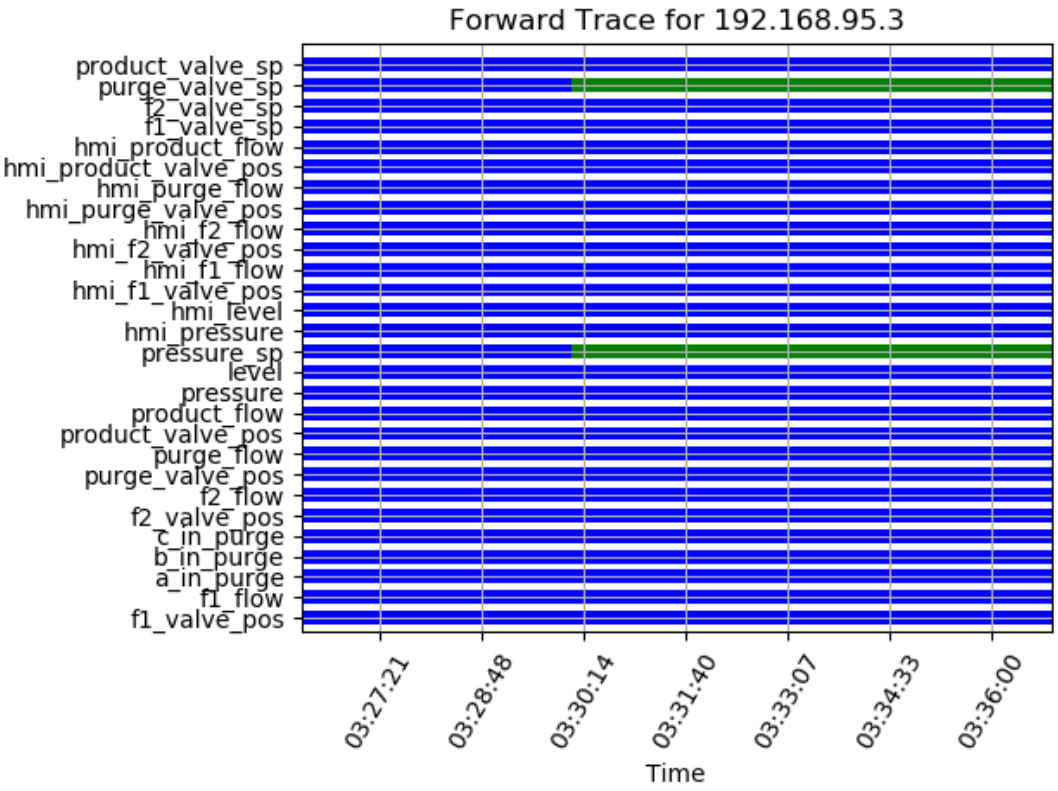

Figure 4.7. Forward trace of all values influenced by the compromised HMI

valve, the sensor reading of the pressure from the tank, and the setting of the pressure setpoint. The first is mostly irrelevant to why the purge valve opened and the second can be sanity-checked for correctness. The investigator can then note that the log entry for the pressure setpoint contains an unreasonable value (zero) and occurs near the same time the purge valve opens. They can thus reasonably conclude that the third log entry must be the cause of the undesirable behavior. The investigator can then continue their investigation at the log entry's source (the compromised HMI).

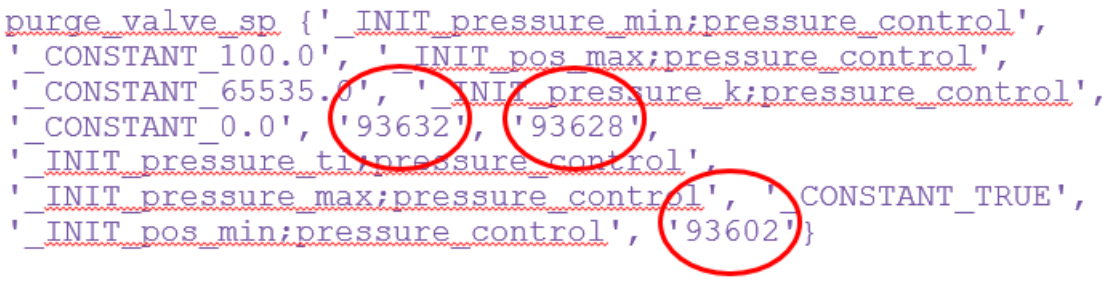

Figure 4.8. The influences on the purge valve setpoint at the moment it opens; log entry ids are circled in red

5 CONCLUSIONS

We introduced a method for simulating the historical execution of a control system. The system is abstracted into a set of variables controlled by one or more controllers, and the behavior of the system is simulated based on some set of logged values. This allows for a complete reconstruction of the historical behavior of the system while limiting needed logging to just the inputs and outputs of the system. The simulation also allows for the tracing of specific input and output values to enable the construction of an end-to-end attack story from the system's perspective.

This has the advantage that the entire range of possible system behavior is accounted for in the model. In contrast, methods that use anomaly detection [22] [8] [9] might generate false positives when observing behavior that is unusual, but still correct, especially if the training set is incomplete or out of date. It also has an advantage over using real physical controllers [19] [18] for emulation in that it is easier to simulate historical data without setup delays, and that the system can be simulated faster than real time, allowing historical data over a long period of time to be analyzed quickly. It also limits necessary logging to just the system's inputs and outputs, removing the need to log every variable in the system separately, which could be prohibitively expensive from a performance standpoint [27].

It also has an advantage over many specification-based systems [14] [25] in that less manual effort is required because the programs of each controller are analyzed directly, inferring the system's intended behavior without relying on expert knowledge. This makes the method easier to generalize between different systems and makes adaptation to a change in system configuration easier to manage.

The method described in this paper was created to assist in forensic investigation, but it could potentially be applied to intrusion detection. In this case, the simulator would be fed (near) real-time input from the process and would verify controller

output as it was produced. The results would then trigger an alert whenever certain conditions were met (e.g. sustained deviation over a threshold for a certain length of time). This would make it more analogous to existing intrusion detection systems. While some additional factors such as the ordering of log entries must be considered in this case, we believe that this is a promising avenue of future research.

One limitation of this method is that it relies on having a clean copy of the programs of each controller. This could be difficult if the program is proprietary information or if an attacker somehow managed to corrupt all saved copies of the program. A related problem is that controller programs can come in many different formats and paradigms, even if we limit ourselves to the industrial control space. While we believe the model is generalizable between different vendors and languages, a manual one-time effort is still needed to support each new type of program.

Nevertheless, we believe that our method is a step forward in control system forensics. It allows investigators to determine what happened over a given time in the system, provides a baseline for what should have happened in the system, and assists in discovering why certain things of interest happened. This aids forensic investigators in quickly investigating an attack on a control system while requiring less manual effort from experts familiar with the system.

REFERENCES

REFERENCES

[1] Sajid Nazir, Shushma Patel, and Dilip Patel. Assessing and augmenting SCADA cyber security: A survey of techniques. *Computers & Security*, 70:436–454, September 2017.

[2] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet Dossier. Technical report, Symantec Corporation, February 2011.

[3] Dragos, Inc. CRASHOVERRIDE: Analyzing the Threat to Electric Grid Operations. Technical report, Dragos, Inc., June 2017.

[4] R. M. van der Knijff. Control systems/SCADA forensics, what's the difference? *Digital Investigation*, 11(3):160–174, September 2014.

[5] Pedro Taveras N. SCADA LIVE FORENSICS: REAL TIME DATA ACQUISITION PROCESS TO DETECT, PREVENT OR EVALUATE CRITICAL SITUATIONS. *European Scientific Journal, ESJ*, 9(21), July 2013.

[6] Keith Stouffer, Suzanne Lightman, Victoria Pillitteri, Marshall Abrams, and Adam Hahn. Guide to Industrial Control Systems (ICS) Security. Technical Report NIST Special Publication (SP) 800-82 Rev. 2, National Institute of Standards and Technology, June 2015.

[7] Chen Markman, Avishai Wool, and Alvaro A. Cardenas. Temporal Phase Shifts in SCADA Networks. In *Proceedings of the 2018 Workshop on Cyber-Physical Systems Security and PrivaCy - CPS-SPC '18*, pages 84–89, Toronto, Canada, 2018. ACM Press.

[8] Marco Caselli, Emmanuele Zambon, and Frank Kargl. Sequence-aware Intrusion Detection in Industrial Control Systems. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security - CPSS '15*, pages 13–24, Singapore, Republic of Singapore, 2015. ACM Press.

[9] V. L. Do, L. Fillatre, and I. Nikiforov. A statistical method for detecting cyber/physical attacks on SCADA systems. In *2014 IEEE Conference on Control Applications (CCA)*, pages 364–369, October 2014.

[10] Dina Hadiosmanovi, Robin Sommer, Emmanuele Zambon, and Pieter H. Hartel. Through the Eye of the PLC: Semantic Security Monitoring for Industrial Processes. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 126–135, New York, NY, USA, 2014. ACM. event-place: New Orleans, Louisiana, USA.

[11] William Jardine, Sylvain Frey, Benjamin Green, and Awais Rashid. SENAMI: Selective Non-Invasive Active Monitoring for ICS Intrusion Detection. In *Proceedings of the 2Nd ACM Workshop on Cyber-Physical Systems Security and*

Privacy, CPS-SPC '16, pages 23–34, New York, NY, USA, 2016. ACM. event-place: Vienna, Austria.

[12] C. Wressnegger, A. Kellner, and K. Rieck. ZOE: Content-Based Anomaly Detection for Industrial Control Systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 127–138, June 2018.

[13] Rafael Ramos Regis Barbosa, Ramin Sadre, and Aiko Pras. Flow whitelisting in SCADA networks. *International Journal of Critical Infrastructure Protection*, 6(3):150–158, December 2013.

[14] Yong Wang, Zhaoyan Xu, Jialong Zhang, Lei Xu, Haopei Wang, and Guofei Gu. SRID: State Relation Based Intrusion Detection for False Data Injection Attacks in SCADA. In *Computer Security - ESORICS 2014*, pages 401–418. Springer, Cham, September 2014.

[15] M. Parvania, G. Koutsandria, V. Muthukumary, S. Peisert, C. McParland, and A. Scaglione. Hybrid Control Network Intrusion Detection Systems for Automated Power Distribution Systems. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 774–779, June 2014.

[16] Davide Fauri, Daniel Ricardo dos Santos, Elisa Costante, Jerry den Hartog, Sandro Etalle, and Stefano Tonetta. From System Specification to Anomaly Detection (and back). In *CPS-SPC@CCS*, 2017.

[17] Dina Hadiosmanovi, Damiano Bolzoni, and Pieter H. Hartel. A log mining approach for process monitoring in SCADA. *International Journal of Information Security*, 11(4):231–251, August 2012.

[18] Lee W. Lerner, Mohammed M. Farag, and Cameron D. Patterson. Run-time prediction and preemption of configuration attacks on embedded process controllers. In *Proceedings of the First International Conference on Security of Internet of Things - SecurIT '12*, pages 135–144, Kollam, India, 2012. ACM Press.

[19] Stephen McLaughlin, Saman Zonouz, Devin Pohly, and Patrick McDaniel. A Trusted Safety Verifier for Process Controller Code. In *Proceedings 2014 Network and Distributed System Security Symposium*, San Diego, CA, 2014. Internet Society.

[20] N. Teja Chiluvuri, Omkar A. Harshe, Cameron D. Patterson, and Cameron T. Baumann. Using Heterogeneous Computing to Implement a Trust Isolated Architecture for Cyber-Physical Control Systems. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security - CPSS '15*, pages 25–35, Singapore, Republic of Singapore, 2015. ACM Press.

[21] Jairo Giraldo, David Urbina, Alvaro Cardenas, Junia Valente, Mustafa Faisal, Justin Ruths, Nils Ole Tippenhauer, Henrik Sandberg, and Richard Candell. A Survey of Physics-Based Attack Detection in Cyber-Physical Systems. *ACM Computing Surveys (CSUR)*, 51(4):76, September 2018.

[22] Hamid Reza Ghaeini, Daniele Antonioli, Ferdinand Brasser, Ahmad-Reza Sadeghi, and Nils Ole Tippenhauer. State-aware anomaly detection for industrial control systems. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing - SAC '18*, pages 1620–1628, Pau, France, 2018. ACM Press.

[23] Y. Mo, R. Chabukswar, and B. Sinopoli. Detecting Integrity Attacks on SCADA Systems. *IEEE Transactions on Control Systems Technology*, 22(4):1396–1407, July 2014.

[24] Marina Krotofil, Jason Larsen, and Dieter Gollmann. The Process Matters: Ensuring Data Veracity in Cyber-Physical Systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security - ASIA CCS '15*, pages 133–144, Singapore, Republic of Singapore, 2015. ACM Press.

[25] Alvaro A. Crdenas, Saurabh Amin, Zong-Syun Lin, Yu-Lun Huang, Chi-Yen Huang, and Shankar Sastry. Attacks Against Process Control Systems: Risk Assessment, Detection, and Response. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 355–366, New York, NY, USA, 2011. ACM. event-place: Hong Kong, China.

[26] David I. Urbina, Jairo A. Giraldo, Alvaro A. Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. Limiting the Impact of Stealthy Attacks on Industrial Control Systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1092–1105, New York, NY, USA, 2016. ACM. event-place: Vienna, Austria.

[27] Andrew Nicholson, Helge Janicke, and Antonio Cau. Safety and Security Monitoring in ICS/SCADA Systems. In *2nd International Symposium for ICS & SCADA Cyber Security Research 2014*. BCS Learning & Development, September 2014.

[28] David Formby, Milad Rad, and Raheem Beyah. Lowering the Barriers to Industrial Control System Security with GRFICS. In *2018 USENIX Workshop on Advances in Security Education (ASE 18)*, Baltimore, MD, 2018. USENIX Association.

[29] openplc | PLCOpen Editor. `https://www.openplcproject.com/plcopen-editor`.

[30] International Electrotechnical Commission. Programmable controllers – Programming languages. Standard, International Electrotechnical Commission, Geneva, CH, February 2013.

[31] E. Estevez, M. Marcos, D. Orive, E. Irisarri, and F. Lopez. Xml based visualization of the iec 61131-3 graphical languages. In *2007 5th IEEE International Conference on Industrial Informatics*, volume 1, pages 279–284, June 2007.

[32] Herbert Prahofer, Roland Schatz, Christian Wirth, and Hanspeter Mossenbock. A Comprehensive Solution for Deterministic Replay Debugging of SoftPLC Applications. *IEEE Transactions on Industrial Informatics*, 7(4):641–651, November 2011.

[33] Unity engine. `https://unity.com/`.

[34] T. R. Alves, M. Buratto, F. M. de Souza, and T. V. Rodrigues. OpenPLC: An open source alternative to automation. In *IEEE Global Humanitarian Technology Conference (GHTC 2014)*, pages 585–589, October 2014.

[35] Qing Liu and Yingmei Li. Modbus/TCP based Network Control System for Water Process in the Firepower Plant. In *2006 6th World Congress on Intelligent Control and Automation*, volume 1, pages 432–435, June 2006.

[36] HMI Software by AdvancedHMI, Application Creation Framework. `https://www.advancedhmi.com/`.