Forensic Investigation of Industrial Control Systems Using Deterministic Replay

Gregory Walkup*, Sriharsha Etigowni†, Dongyan Xu†, Vincent Urias*, and Han W. Lin*
* *Sandia National Laboratories*, Albuquerque, USA
Email: {gwalkup, veuria, hwlin}@sandia.gov
† *Purdue University*, West Lafayette, USA
Email: {setigown, dxu}@purdue.edu

***Abstract*—From manufacturing plants to power grids, industrial control systems are increasingly controlled and networked digitally. While networking these systems together improves their efficiency and convenience to control, it also opens them up to attacks by malicious actors. When these attacks occur, forensic investigators should be able to determine what was compromised and which corrective actions need to be taken.**

In this paper, we propose a method to investigate attacks on industrial control systems by simulating the logged inputs of the system over time using a model constructed from the control programs. We detect any attacks that will lead to perturbations of the normal operation of the system by comparing the simulated output to the actual output. We also perform dependency tracing between the inputs and outputs of the system, so that attacks can be traced from the anomaly to their sources and vice-versa. Our method can greatly aid investigators in recovering the complete attack graph used by the attacker using only the input and output logs from an industrial control system. To evaluate our method, we constructed a hybrid testbed with a simulated version of the Simplified Tennessee Eastman process, using a hardware-in-the-loop Allen-Bradley Micrologix 1100 PLC. We were able to accurately detect all attack anomalies with a false positive rate of 0.3% or less.

***Index Terms*—Industrial Control Systems, PLC**

I. Introduction

Attacks such as Stuxnet [1] and Crashoverride [2] show that attackers are actively targeting industrial control systems. It is therefore important that the entities which use such systems are able to detect and investigate these attacks. However, due to the wide variety of equipment and configurations in industrial control systems, it is difficult for forensic experts to gather enough domain specific knowledge to quickly and accurately investigate a particular attack [3]. Furthermore, shutting down a control system while an incident is being investigated may be impossible or cost large amounts of money in lost revenue [4].

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 be clear that an attack has occurred, or what its effects were [5]. If it is determined that an attack did occur, investigators need to find the root cause of the failure and trace it back to a physical input or a traditional IT system to continue the investigation through more traditional methods. If investigators can detect an attacker's changes to the normal operation of the system, it is far easier for investigators to trace the history of the attack and find the cause of the failure. To do this, the investigators first need to establish what the baseline behavior of the system is in order to see what the attacker has changed.

In this work, we propose a new method of investigating attacks on industrial control systems by simulating the system on its actual historical inputs. If all legitimate inputs to the system are logged and we can obtain a behavior model of the automated components of the system, then simulating that model over the historical inputs gives a baseline for what "*should have happened*" in the system at that time. From that baseline, we can extract the expected system outputs at each point in time and compare them to the actual output of the system at that time using the logged data. This aids investigators by allowing them to see if any outputs of the system were unexpected, given the system's inputs. From there, they can examine the production of those outputs more closely in order to determine if malicious activity occurred. For example, if a controller suddenly starts producing output that is out of line with the input it receives, that is an indication that it may have been compromised by a malicious adversary.

However, sometimes this solution is not sufficient. An attacker can disguise their actions as legitimate input and allow them to be logged and fed into the simulation. This means that the simulation would include the attacker's changes and not differ from the historical execution of the system. Since the simulator cannot a priori distinguish illegitimate input to the system, we instead provide investigators with the ability to trace data as it moves through the system. By connecting each input with its dependent outputs (and vice-versa), investigators are able to trace an undesirable outcome to all inputs which affected it or an undesirable input to all outputs it affects. This helps investigators narrow the focus of their search and reach a conclusion more efficiently. This provides information on *why* something happened even if the system did not behave anomalously.

This investigation framework can thus provide investigators with knowledge of *i)* what happened in the system, *ii)* what should have happened in the system, and *iii)* why something happened in the system. Significantly, the framework also aims to do this while requiring no invasive modifications to the system's components and minimizing the amount of data that needs to be logged.

II. Background and Overview

In this section, we will provide background required for our paper on industrial control systems, describe the attacker model, the overview of our solution, and our system abstraction.

A. Industrial Control Systems

At a low level, industrial control systems are usually composed of actuators and sensors which are connected to one or more remote terminal units (RTUs) and/or programmable logic controllers (PLCs). Typically, RTUs have no logic of their own and merely collect data from sensors and forward user commands to actuators, whereas a PLC is capable of making its own decisions based on sensor data. Human-machine interfaces (HMIs) allow an operator to monitor parts of the system and issue commands as necessary. Many industrial control systems have components which reside on traditional IT systems for command, control, and logging (such as a data historian or a web server), but these high-level devices are not essential and will not be explicitly modeled in this work.

B. 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 (e.g. by launching an attack from inside a trusted perimeter), we assume that an attacker does not have access to the defense solution system and is not able to arbitrarily manipulate or delete log entries which reside on this system. We assume that our defense solution system is trusted and is isolated from the control system network. 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 the reference code to redefine the system's behavior. We believe that this is a reasonable assumption, given that a model for the system behavior must originally come from somewhere.

C. Overview

When an industrial control system does not behave correctly, forensic investigators are often tasked with discovering why something went wrong. However, industrial control systems may produce huge amounts of log data, and even if it can be effectively visualized, it may be difficult to connect cause and effect without substantial amounts of domain knowledge about the specific system being investigated. This makes it difficult to find anomalies for further investigation.

Fig. 1. Example log format

Our goal is to assist investigators by helping them quickly determine the root cause in an industrial control system and to provide further points of interest at which the investigation can continue. Ideally, we would like to do this in a way that minimizes the need to consult experts on the system in question. Our solution provides the point of interest from where the investigation can be started, so that investigators do not have to spend time manually understanding and analyzing the system to determine the source of the problem.

Our main aim is to provide a non intrusive solution with minimal to no overhead. Hence, any logging overhead incurred to support forensics should also be non-intrusive and should not affect the normal operation of the system being logged. The system itself and any constituent programs should also not need to be modified in any significant way.

D. System Abstraction

Control System Abstraction We have abstracted an industrial control system as a collection of control, configuration, and state variables for our purposes. The control variables are user inputs, inputs from the sensors, and outputs to the actuators. The configuration variables are the configuration settings for the control system and the state variables represent the system state. The abstraction 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 was inspired by the previous approaches in literature [6], [7], [8].

Control Program A *control program* is an sequence of operations running on a controller. A control program can manipulate controller variables and can be run continuously, on a timed loop, or in response to a particular variable. The control program(s) in a controller represent the automation logic present within that controller's real-world analogue.

Controller All control variables in an industrial control system reside in a *controller*. The controller is responsible for manipulating the variables or using them to make decisions. In the real world, controllers correspond to PLCs, RTUs, 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.

Log Entries Fundamentally, each remote action taken in the system must consist of *i)* the requesting device, *ii)* the requestee device, and *iii)* the data being transferred. Most ICS protocols also include a fourth field which is the addressable location of the data in question. Figure 1 shows an example of the information collected by logging these actions. 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.

III. System Design

In this section we describe our system design, including the logging of variables and important data, the construction of a model containing those variables, our method of simulation with deterministic replay, and forensic investigation to find the root cause.

A. Logging Variables

Modern industrial control systems often interact 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 [7]. This allows for log collection that is unobtrusive, does not significantly impact the performance of the system, and is often supported by commercial monitoring tools. 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 [9].

The 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. 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.

B. Model Construction

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 such as a function block diagram or ladder logic, as defined by IEC 61131-3 [10] (the IEC standard for programming languages in PLCs). In our evaluation system, we extract control programs out of a given ladder logic diagram. Different vendors and devices can use a variety of different program formats, so a *one-time manual effort* would have to be made to ingest programs from a new type of device.

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 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.

C. Discrete Event Simulation

Algorithm 1: Event Simulator Replay Algorithm

```
Data: one or more log files; one or more controller programs
Result: a collection of historical variable values
initialize event calendar eventCal;
foreach input log file lf do
    place lf's first event into eventCal at its logged time;
end
foreach controller program prog do
    parse prog into a list of statements;
    initialize all variables used by prog;
    extract execution interval interval from prog;
    place a prog execution event in eventCal interval time
      after the earliest event;
end
create a variable history for each defined variable;
while eventCal contains at least one log event do
    get event event from eventCal with the earliest time;
    set currTime to be event's time;
    if event is a log entry event then
        // For each variable being accessed in
            entry
        foreach (var, value) in entry do
            if entry shows data flowing to controller then
                set var to value and var's trace to entry's id;
                record [currTime, value, trace] in var's
                  variable history;
            end
            else if entry shows data flowing from controller then
                get var's value and trace, simVal and trace;
                record [currTime, value, simVal, trace] in
                  var's variable history;
            end
        end
        place next log entry (if any) from associated log file into
          eventCal at its logged time;
    end
    else if event is a program execution event then
        extract (prog, interval) from event;
        // See algorithm 2 for program execution
        execute prog and update affected variables;
        place another prog execution event in eventCal at time
          currTime + interval;
    end
end
return each non-empty variable history;
```

After the logs are ingested and the system model constructed, the entire system is simulated in a discrete event simulator. This process is shown in algorithm 1. 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.

Algorithm 2: Controller Program Simulation

```
Data: variables used by the called function; a list executionList
      of function call statements which compose the program
Result: the function's outputs
initialize local variables from arguments;
foreach statement e in executionList do
    collect e's arguments from local or global variables;
    if e is a controller builtin function then
        execute e using the predefined function;
    else
        recursively execute e;
    end
    collect outputs from executed function;
    copy outputs and their traces into variables specified in e;
end
collect results from local variables;
return results;
```

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. This process is shown in algorithm 2. 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. A new execution event is also generated in the future after the appropriate interval if applicable.

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. 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 [11] when the inputs and outputs of the system are logged in their entirety. This high degree of determinism makes this simulation approach more feasible in industrial control systems than it would be in traditional computing systems.

D. Forensic 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: *i)* control program-based attacks and *ii)* data-based attacks.

1) Control Program-Based Attacks: These kind of attacks alter the control program 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 an "*expected*" state, a "*warning*" state, or an "*error*" state. These thresholds are configurable; in our evaluation, 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.

2) Data-Based Attacks: These kind of attacks tamper 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 [12], [13], [14].

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.

IV. Evaluation

We evaluated our approach using a modified version of the the GRFICS [15] graphical interface. The GRFICS graphical portion is designed to simulate the Simplified Tennessee Eastman process, in which reactants mix together into a tank to form a liquid product. Instead of running GRFICS with its default setup, we connected the simulation interface to a Micrologix 1100, a PLC manufactured by Allen-Bradley. In this way, we were able to construct a "*hardware-in-the-loop*" setup in which a physical PLC controlled the GRFICS simulation. This allowed us to construct a realistic testbed without having to build a physical process to be controlled.

A. Experimental Setup

Our test setup consisted of two machines networked with the physical PLC device on a local area network. One machine ran a simulation of the physical process and the other ran an HMI (human-machine interface) that monitored the status of the simulated process. The HMI was run on a Windows 10 machine, while the simulation was run on a VM running Ubuntu 18.04. The second machine with the HMI was also used to sniff and log network packets. The basic configurations of both machines were taken from GRFICS, but were modified to accommodate using the physical PLC.

The simulation machine includes a graphical component written in the Unity Engine [16] that shows the current physical status of the plant. Both the simulation itself and the graphical component were taken from GRFICS [15], but the scripts that handled communication were modified to accommodate communication with the original PLC using the Ethernet/IP [17] protocol instead of Modbus over TCP [18]. The second machine ran the GRFICS HMI built in Advanced HMI [19] with the communications modified in a similar way.

All tests were performed by first starting the logging mechanism, which passively sniffed packets traveling over the shared LAN used by all devices in the system. Logging was performed by a custom python script that used pyshark [20] (a python interface for Wireshark) to sniff network traffic that appeared to be Ethernet/IP traffic. This sniffed data was converted to the log format used by the simulator in real time and stored locally. After logging was started, the graphical interface for the emulated system was started and the PLC was placed into "*run*" mode, starting the program. Finally, the HMI program was started, simulating an operator monitoring the process. 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.

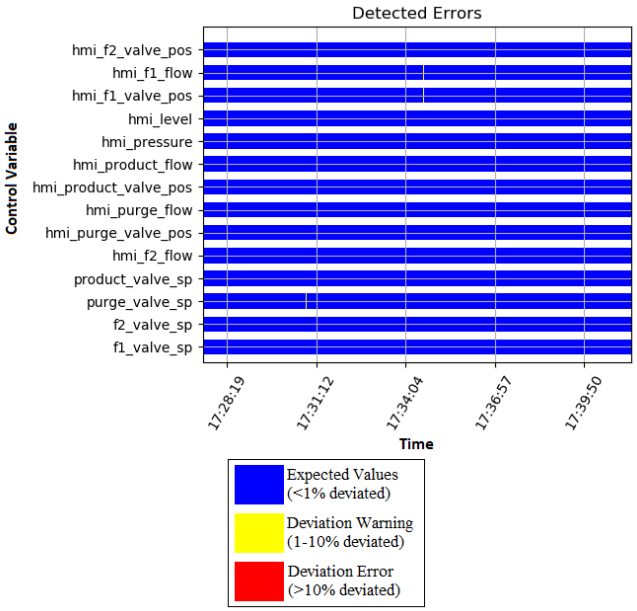

Fig. 2. Error Timeline for Normal Operation

TABLE I
Simulator Trial Results

Trial	Log Entries	Accuracy	Simulation Time (s)
Normal Operation	40060	99.9%	72
Program Attack	55379	99.8%	135
Data Attack	50881	99.7%	155

After the system was run and logging data collected, the simulator was run over the collected data. For our specific setup, a configuration file was created detailing the IP address of the PLC and some information about its starting state and cosmetic settings for the output. After being fed the PLC program and the log file, the simulator was run to produce error data for each variable, which was then graphed. Each simulation run was performed using an Intel i7 5700 CPU. The results are detailed in each subsection below. Statistics from each run are shown in table I. While the attack simulations took longer because of the longer physical running time of the trial (with the data attack also having some additional tracing overhead), all simulations were able to run substantially faster than real time.

B. Normal Operation

This test was performed to demonstrate the accuracy of the simulator in simulating the normal behavior of the system. The system was run normally for about 10 minutes with no attack

being performed. 58 log entries were flagged as warnings (more than 1% deviated, but less than 10%) and 2 log entries were flagged as errors (more than 10% deviated), leading to a total false positive rate of 0.1%.

Figure 2 shows the number of errors detected in the system over time for each variable being monitored in the system. Blue sections represent values that were less than 1% deviated from expected values (when compared to the total range of observed values), yellow sections represent values that were between 1% and 10% deviated, and red sections represent values that were more than 10% deviated. Variable names are to the right of each bar. As no attack was performed, most of the graph is blue, with some very short periods of small deviations for some variables.

In general, most detection errors stem from places where outputs change very quickly. An example is when the purge valve rapidly moves from fully closed to fully open around the 17:30 mark of the test. In such 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 differing default values when the controller first starts (before either the simulated or the actual controller executes one program cycle). Those errors go away once the simulation and controller have had a chance to execute on the same data.

Overall, the simulator generated very few false positives while being run on the normal operation log data. Each false positive represents one log entry (out of 40k) that was flagged as being different from expectations. In isolation, a single anomalous log entry (or even just a few) is unlikely to indicate an actual malicious event. 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.

C. Program Modification Attack

The system was run normally (as in the first test) for five minutes, after which the control program on the PLC was modified maliciously to increase the pressure in the tank to dangerous levels. 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. Figure 3 shows the number of errors detected in the system over time during the program modification attack. The attack itself begins at timestamp 19:26:24. The sustained red bars clearly show that some of the controller's outputs are deviating form its expected programming when the attack begins. Throughout the attack, the simulator correctly flags all deviating values that the attacker has modified.

Note that in this case, while the fake HMI values and the product valve are quickly marked as erroneous when the attack begins, other logged outputs take some time to diverge

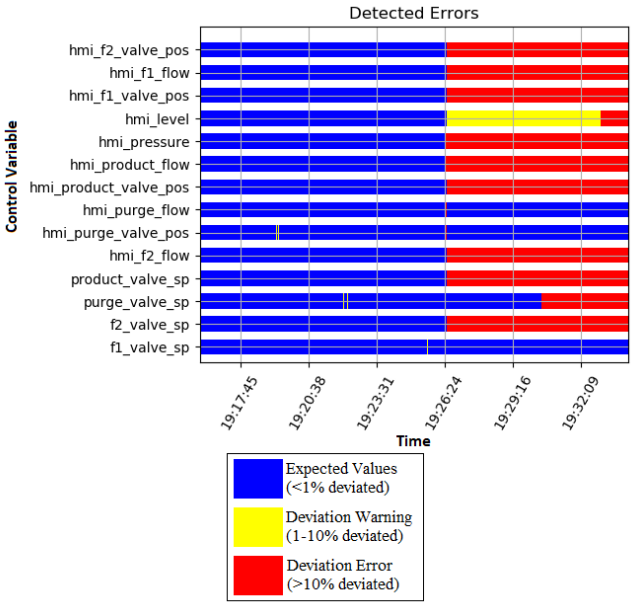

Fig. 3. Error Timeline for Program Modification Attack

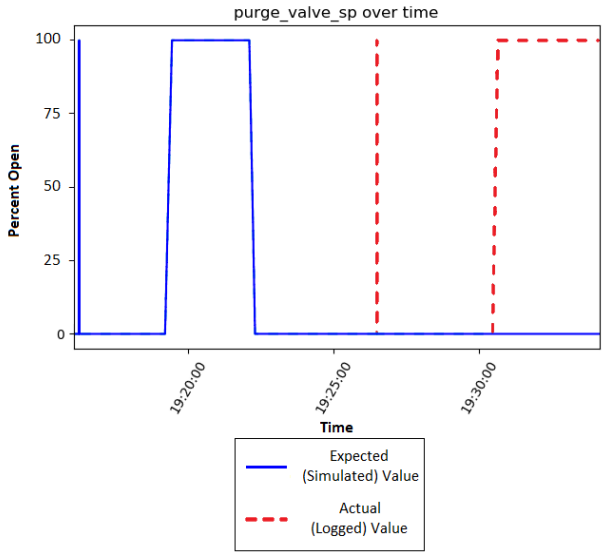

Fig. 4. Actual logged value of the purge valve setpoint (solid line) vs. the simulated value (dashed line)

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, the outputs that do deviate are clearly suspicious because they are a sustained deviation from normal values over a long period of time.

Figure 4 shows a comparison between the actual value of the purge valve as logged (solid blue line) versus the expected value based on the simulation (dashed red line). The values of this particular variable range from 0 when the valve is fully closed to 2^{16} when the valve is fully open (vertical axis labeled

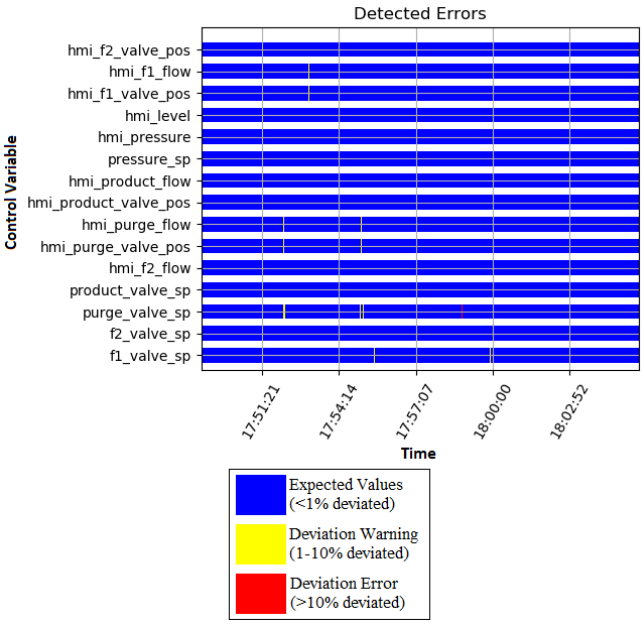

Fig. 5. Error timeline for stealthy program data attack (virtually identical to benign case)

with percentages for clarity). The purge valve is normally supposed to open when the pressure in the tank reaches a certain threshold to relieve pressure. *Early in the graph, the dashed line is covered by the solid blue line, indicating that the simulated and actual values are very close to each other.*

Note that the purge valve (correctly) opens early in the run as part of normal operation, but fails to open as simulated towards the end of the run. This is because the attacker's new program differs from the simulated program in that the purge valve will never open, thus leading to an explosion. The simulated purge valve also briefly opens in the middle of the run due to the simulated controller recording values being written the controller as part of the attacker's program upload. While this makes the detection of the time of program upload a little easier, it would not be considered a meaningful or reliable deviation. The later deviation, which is sustained and shows the influence of the attacker in the operation of the purge valve, would be detected even if the malicious program upload was not logged.

D. Data Modification 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.

Figure 5 shows that, unlike the previous attack, the simulator did not notice any incorrect behavior from the controller. This is because the system was still functioning correctly; in this case the attacker is using a seemingly-valid command from the (compromised) HMI to accomplish their goals. The simulator saw the command from the HMI in the logs and saw that the controller reacted appropriately to the command, so reality ended up perfectly matching the simulation.

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.

Suppose an investigator knows that the HMI was compromised during a specific window of time. They would then reasonably want to know what effect the HMI had on the system during that period. Given the IP address of the HMI as input, the simulator is able to separately note each log entry that involves the HMI and track the variables it modifies. In this case, a value of 0 is written to the variable *pressure_sp* (the target pressure setpoint of the tank) around timestamp 17:57:30. The simulator then notes that *pressure_sp* is involved in a calculation that sets *purge_valve_sp* (the output setpoint of the purge valve). The investigator now knows that the HMI's modification of *pressure_sp* had a secondary effect on the value of *purge_valve_sp*, even if they do not know the particulars of how the system is implemented.

```
purge_valve_sp {'_INIT_pressure_min;pressure_control',
'_CONSTANT_100.0', '_INIT_pos_max;pressure_control',
'_CONSTANT_65535.0', '_INIT_pressure_k;pressure_control',
'_CONSTANT_0.0', '93632', '93628',
'_INIT_pressure_ti;pressure_control',
'_INIT_pressure_max;pressure_control', '_CONSTANT_TRUE',
'_INIT_pos_min;pressure_control', '93602'}
```

Fig. 6. Sources of data related to the purge valve at the moment it opens; log entry ids are circled in red

Similarly, the simulator can trace backwards from an end effect to its causes to aid an investigation. Suppose that in this scenario, someone notices the purge valve is open too often. An investigator can query the simulator about all influences on the position of the purge valve at the time that it opens. Figure 6 shows an example output of the simulator, which shows all influences on the purge valve position at the given time. All values that begin with underscores are internal system values and aren't relevant to the investigation. The three numbers circled in red, however, are log entry ids that correspond to the log entries of inputs that affected this output. The investigator can then narrow his search to these three log entries (out of tens of thousands) and notice that the HMI writes a value of 0 to the variable *pressure_sp*. Since this is a strange value (a desired pressure of 0 is unlikely to be legitimate), the investigator can reasonably conclude that the HMI is the source of the strange behavior and continue his investigation there, where they will discover evidence of the attacker's intrusion.

V. Related Work

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 [21], [22], [13] 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. Our method eliminates the need to collect large amounts of training data at the beginning and any time the system's configuration changes. Knowing the complete specification of the system also accounts for rare scenarios and inputs that may not be captured in training data.

Hadziosmanovic et al. [7] 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 [9], looking for suspicious sequences of events [22], and by decoding even unknown control system protocols [23]. Barbosa et al. [24] also propose a method for whitelisting flows of data rather than sniffing packets for individual variables. Our work uses similar data collection methods, but seeks to detect anomalies based on a system specification rather than statistics.

Other approaches focus on creating a specification for how the system is supposed to operate using expert knowledge. 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. Our method seeks to partially automate the construction of a system model in order to allow forensic investigators who may not be familiar with the system to quickly determine the cause of a problem.

Wang et al.[6] 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 [25] propose using redundant controllers with identical programming in place of a formal model of the system. Fauri et al. [26] 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. [27] 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. Our method expands on this work by introducing a simulation of the system and using that to validate the system's behavior against its specification. This also allows for a more precise reconstruction of historical events using log data.

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.[28] 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.[29] and Do et al. [13] both use Cumulative Sum (CUSUM) to detect anomalies in the physical process itself. Mo et al.[14] and Krotofil et al.[30] describe systems for specifically detecting falsified sensor data. Our method can be seen as complementary to these methods, as it focuses on validating control system behavior rather than the physical process being controller.

VI. Discussion

The method described in this paper was created to assist 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. Future work would likely include testing on a wider variety of hardware, and on more complicated systems, especially those that are actually used in the real world. This would more clearly demonstrate the method's advantages over trying to make sense of the data manually.

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.

VII. Conclusion

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 and not requiring any instrumentation of the logged system. We constructed a hybrid testbed with a simulated version of the Simplified Tennessee Eastman process connected to a Micrologix 1100 PLC as hardware-in-the-loop to evaluate the solution. The solution was able to detect all the times a controller deviated from its expected behavior and allow for tracing of named outputs and inputs while signaling a false positive on less than 0.3% of log entries on average.

VIII. Acknowledgments

We thank the anonymous reviewers for their constructive comments. Xu's effort was supported in part by an LDRD Grant from Sandia National Laboratories. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] Nicolas Falliere, Liam O Murchu, and Eric Chien, "W32.Stuxnet Dossier," Symantec Corporation, Tech. Rep., Feb. 2011. [Online]. Available: https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[2] Dragos, Inc., "CRASHOVERRIDE: Analyzing the Threat to Electric Grid Operations," Dragos, Inc., Tech. Rep., Jun. 2017. [Online]. Available: https://dragos.com/wp-content/uploads/CrashOverride-01.pdf

[3] R. M. van der Knijff, "Control systems/SCADA forensics, what's the difference?" *Digital Investigation*, vol. 11, no. 3, pp. 160–174, Sep. 2014.

[4] Pedro Taveras N., "SCADA LIVE FORENSICS: REAL TIME DATA ACQUISITION PROCESS TO DETECT, PREVENT OR EVALUATE CRITICAL SITUATIONS," *European Scientific Journal, ESJ*, vol. 9, no. 21, Jul. 2013.

[5] K. Stouffer, S. Lightman, V. Pillitteri, M. Abrams, and A. Hahn, "Guide to Industrial Control Systems (ICS) Security," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication (SP) 800-82 Rev. 2, Jun. 2015. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-82/rev-2/final

[6] Y. Wang, Z. Xu, J. Zhang, L. Xu, H. Wang, and G. Gu, "SRID: State Relation Based Intrusion Detection for False Data Injection Attacks in SCADA," in *Computer Security - ESORICS 2014*. Springer, Cham, Sep. 2014, pp. 401–418.

[7] D. Hadžiosmanović, R. Sommer, E. Zambon, and P. H. Hartel, "Through the Eye of the PLC: Semantic Security Monitoring for Industrial Processes," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 126–135, event-place: New Orleans, Louisiana, USA.

[8] A. A. Cárdenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry, "Attacks Against Process Control Systems: Risk Assessment, Detection, and Response," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 355–366, event-place: Hong Kong, China.

[9] W. Jardine, S. Frey, B. Green, and A. 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*, ser. CPS-SPC '16. New York, NY, USA: ACM, 2016, pp. 23–34, event-place: Vienna, Austria.

[10] I. E. Commission, "Programmable controllers – Programming languages," International Electrotechnical Commission, Geneva, CH, Standard, Feb. 2013.

[11] H. Prahofer, R. Schatz, C. Wirth, and H. Mossenbock, "A Comprehensive Solution for Deterministic Replay Debugging of SoftPLC Applications," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 641–651, Nov. 2011.

[12] D. I. Urbina, J. A. Giraldo, A. A. Cardenas, N. O. Tippenhauer, J. Valente, M. Faisal, J. Ruths, R. Candell, and H. Sandberg, "Limiting the Impact of Stealthy Attacks on Industrial Control Systems," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1092–1105, event-place: Vienna, Austria.

[13] 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)*, Oct. 2014, pp. 364–369.

[14] Y. Mo, R. Chabukswar, and B. Sinopoli, "Detecting Integrity Attacks on SCADA Systems," *IEEE Transactions on Control Systems Technology*, vol. 22, no. 4, pp. 1396–1407, Jul. 2014.

[15] D. Formby, M. Rad, and R. Beyah, "Lowering the Barriers to Industrial Control System Security with GRFICS," in *2018 USENIX Workshop on Advances in Security Education (ASE 18)*. Baltimore, MD: USENIX Association, 2018.

[16] U. Technologies. (2019) Unity engine. [Online]. Available: https://unity.com/

[17] ODVA, "EtherNet/IP™ – CIP on Ethernet Technology," Mar. 2016. [Online]. Available: https://www.odva.org/Portals/0/Library/Publications_Numbered/PUB00138R6_Tech-Series-EtherNetIP.pdf

[18] Modbus Organization, Inc., "MODBUS MESSAGING ON TCP/IP IMPLEMENTATION GUIDE V1.0b," Oct. 2006. [Online]. Available: http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf

[19] AdvancedHMI, "HMI Software by AdvancedHMI, Application Creation Framework," https://www.advancedhmi.com/, 2019. [Online]. Available: https://www.advancedhmi.com/

[20] Dor Green (kiminewt), "PyShark." [Online]. Available: https://kiminewt.github.io/pyshark/

[21] C. Markman, A. Wool, and A. A. Cardenas, "Temporal Phase Shifts in SCADA Networks," in *Proceedings of the 2018 Workshop on Cyber-Physical Systems Security and PrivaCy - CPS-SPC '18*. Toronto, Canada: ACM Press, 2018, pp. 84–89.

[22] M. Caselli, E. Zambon, and F. Kargl, "Sequence-aware Intrusion Detection in Industrial Control Systems," in *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security - CPSS '15*. Singapore, Republic of Singapore: ACM Press, 2015, pp. 13–24.

[23] 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)*, Jun. 2018, pp. 127–138.

[24] R. R. R. Barbosa, R. Sadre, and A. Pras, "Flow whitelisting in SCADA networks," *International Journal of Critical Infrastructure Protection*, vol. 6, no. 3, pp. 150–158, Dec. 2013.

[25] 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*, Jun. 2014, pp. 774–779.

[26] D. Fauri, D. R. d. Santos, E. Costante, J. d. Hartog, S. Etalle, and S. Tonetta, "From System Specification to Anomaly Detection (and back)," in *CPS-SPC@CCS*, 2017.

[27] D. Hadžiosmanović, D. Bolzoni, and P. H. Hartel, "A log mining approach for process monitoring in SCADA," *International Journal of Information Security*, vol. 11, no. 4, pp. 231–251, Aug. 2012.

[28] J. Giraldo, D. Urbina, A. Cardenas, J. Valente, M. Faisal, J. Ruths, N. O. Tippenhauer, H. Sandberg, and R. Candell, "A Survey of Physics-Based Attack Detection in Cyber-Physical Systems," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 76, Sep. 2018.

[29] H. R. Ghaeini, D. Antonioli, F. Brasser, A.-R. Sadeghi, and N. O. Tippenhauer, "State-aware anomaly detection for industrial control systems," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing - SAC '18*. Pau, France: ACM Press, 2018, pp. 1620–1628.

[30] M. Krotofil, J. Larsen, and D. 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*. Singapore, Republic of Singapore: ACM Press, 2015, pp. 133–144.