SABOT: Specification-based Payload Generation for Programmable Logic Controllers*

Stephen McLaughlin Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory
Pennsylvania State University
{smclaugh,mcdaniel}@cse.psu.edu

ABSTRACT

Programmable Logic Controllers (PLCs) drive the behavior of industrial control systems according to uploaded programs. It is now known that PLCs are vulnerable to the uploading of malicious code that can have severe physical consequences. What is not understood is whether an adversary with no knowledge of the PLC's interface to the control system can execute a damaging, targeted, or stealthy attack against a control system using the PLC. In this paper, we present SABOT, a tool that automatically maps the control instructions in a PLC to an adversary-provided specification of the target control system's behavior. This mapping recovers sufficient semantics of the PLC's internal layout to instantiate arbitrary malicious controller code. This lowers the prerequisite knowledge needed to tailor an attack to a control system. SABOT uses an incremental model checking algorithm to map a few plant devices at a time, until a mapping is found for all adversary-specified devices. At this point, a malicious payload can be compiled and uploaded to the PLC. Our evaluation shows that SABOT correctly compiles payloads for all tested control systems when the adversary correctly specifies full system behavior, and for 4 out of 5 systems in most cases where there where unspecified features. Furthermore, SABOT completed all analyses in under 2 minutes.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; J.7 [**Computers in Other Systems**]: Process control

General Terms

Security

Keywords

Programmable Logic Controller, Attack, Critical Infrastructure

*This material is based upon work partially supported by the National Science Foundation under Grants CCF 0937944 and CNS 0643907, and by a grant from the Security and Software Engineering Research Center (S^2ERC).

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

1. INTRODUCTION

Process control systems are vulnerable to software-based exploits with physical consequences [16, 33, 27, 35, 37, 3]. The increased network connectivity and standardization of Supervisory Control and Data Acquisition (SCADA) have raised concerns of attacks on control system-managed infrastructure [5, 20, 6, 44]. Control systems use Programmable Logic Controllers (PLCs) to drive physical machinery according to software control logic. For ease of modification, control logic is uploaded to the PLC from the local network, the Internet, or serial port [41, 11]. For example, hundreds of Internet accessible PLCs and SCADA devices can be found through the Shodan search engine [1, 12]. An adversary with PLC access can upload malicious control logic, called the *payload*, thereby gaining full control of the devices under the PLC. Nevertheless, an additional challenge still remains: *Even with knowledge of the target control system's behavior, an adversary cannot construct a payload for specific devices without knowing how the PLC interfaces with those devices.* However, there is often no direct way of determining which PLC memory locations regulate which devices.

This paper presents SABOT (Specification-based Attacks against Boolean Operations and Timers)[1], a proof-of-concept tool for generating PLC payloads based on an adversary-provided specification of device behavior in the target control system. SABOT's main purpose is to recover semantics of PLC memory locations that are mapped to physical devices. Specifically, SABOT determines which PLC memory locations map to the devices in a behavioral specification of the target plant. This enables an adversary with a goal to attack specific devices to automatically instantiate that attack for a specific PLC in the victim control system. Unlike previous attacks, such as Stuxnet [16], a precompiled payload is not necessary.

An attack using SABOT proceeds as follows:

1. The adversary encodes his understanding of the plant's behavior into a specification. The specification contains a declaration of plant devices and a list of temporal logic properties defining their behavior.
2. SABOT downloads the existing control logic bytecode from the victim PLC, and decompiles it into a logical model. SABOT then uses model checking to find a mapping between the specified devices and variables within the control logic.
3. SABOT uses the mapping to instantiate a generic malicious payload into one that can be run on the victim PLC. The generic payload can contain arbitrary manipulation of the specified devices, which SABOT substitutes with PLC addresses.

[1]Sabots were wooden shoes thrown into the gears of machinery by Luddites in the Industrial Revolution.

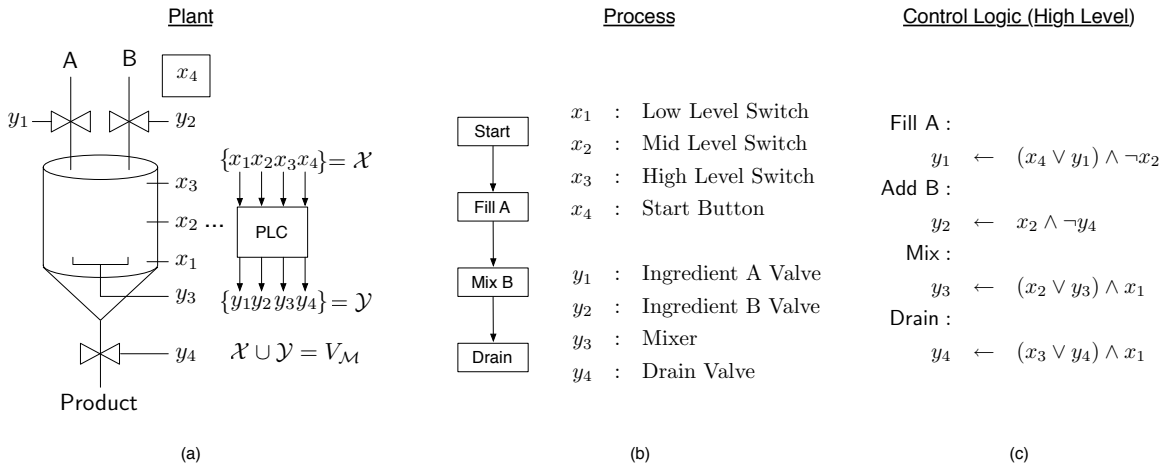


Figure 1: An example plant and process specification for a chemical mixing control process.

SABOT is not for adversaries that do not understand the behavior of the victim plant. In such cases, an adversary can erase the PLC's memory, upload random instructions, or attempt to bypass safety properties of the control logic [29]. None of these attacks, however, are guaranteed to be effective or stealthy. Instead, SABOT is most useful to adversaries with accurate knowledge of the target plant and process, but that are unaware of a critical piece of information: *how to manipulate specific plant devices from the PLC*. This information is not obvious to the adversary because it is unknown which PLC variables are mapped to plant devices.

There are several implications stemming from the existence of a tool like SABOT:

1. Reduced adversary requirements. In most cases, the only way to know which PLC memory addresses map to which devices is to physically inspect the labels on wires connecting the PLC to devices. Only the most powerful adversaries e.g., insiders or nation states, will have this information [38]. Furthermore, SABOT enables attacks by adversaries that are unfamiliar with PLC instruction set architectures and communication protocols. Because the variable mapping is done on an intermediate representation, code from arbitrary PLCs can be decompiled for analysis by SABOT. We give an example implementation of one such decompiler in Section 3.2.

2. Improved target identification. In the Stuxnet attack, PLC version strings and device metadata were used to verify that the correct target had been identified [16]. If this metadata was not found in the PLC, then it was silently ignored by the virus. Using SABOT, a target is identified by whether or not its control logic behaves in the way specified by the adversary. Thus, the adversary need not know any version strings or vendor metadata *a priori*. This may also reduce false positives in cases where an unintended PLC contains the expected version strings and metadata. We evaluate SABOT's ability to correctly identify a target control logic out of a number of candidates in Section 4.4.

3. Room for error. Using a precompiled payload, an adversary's understanding of the plant's behavior must be exactly correct or the payload will likely fail. Using a dynamically generated payload, there is some room for error. For example in Section 4.5 we describe a method for writing behavioral specifications that will correctly map to a control logic regardless of whether or not it implements an emergency stop button, a common feature in many control systems. While this is a first step towards making a truly adaptable attack mechanism, our results show that it works well when an adversary correctly specifies the behavior of a majority of plant devices, as shown in Section 4.1. It is also, to the best of our knowledge, the first analysis of its kind directed at PLC-based control systems.

Our evaluations of SABOT show that it performs accurately and efficiently against control logics of equal or greater complexity than the target of the Stuxnet attack [16]. We begin our discussion of SABOT in the next section by providing a brief overview of sequential control systems and detail an illustrative example.

2. CONTROL SYSTEMS

Control systems are used to monitor and control physical processes. These systems can drive processes as simple as motion activated light switches or as complex as wastewater treatment. Regardless of their purpose and complexity, control systems are generally structured the same.

The physical apparatus in which the control system resides is called the *plant*. Within the plant, control systems can be decomposed into three distinct elements: *control inputs*, *control outputs*, and *control logic*. Control inputs are used to communicate the state of the plant. For example, temperature, motion, or light sensors are used to detect and communicate physical states. Other inputs are human driven, such as switches, dials, and buttons. The control system sends output signals to external devices to effect changes in the physical world. For example, such signals may turn on indicator lights, open and close valves, or drive bi-directional motors. The control logic is the PLC software that computes new outputs based on sensor inputs. The PLC repeatedly executes the control logic in a *scan cycle* consisting of: *i.* read new inputs, *ii.* execute control logic, and *iii.* write outputs to devices.

An example control system for a simplified chemical mixer is shown in Figure 1. The plant (Figure 1(a)) is a single mixer with valves to dispense two ingredients, A and B, a mixing element, and a valve for draining the tank. The valves are controlled by the output variables y_1 and y_2 respectively, the mixer by y_3, and the drain by y_4. A device is ON when its corresponding output variable is set to $\top$ (true) and OFF when it is $\bot$ (false). Three level sensors are used to detect when the tank is at three levels–low, half full, and full (corresponding to inputs x_1, x_2, and x_3, respectively). A level switch $x_i = \top$ if the contents of the tank are at or above its level. A start signal is sent to the PLC via x_4.

The mixer follows a simple process in which ingredient A is added until the tank is half full, ingredient B is mixed with A until the tank is full, and the result is drained. The specification, shown in Figure 1(b), details the control system implementation:

1. Initially, all inputs and outputs are off until the Start button is pressed ($x_4 = \top$).
2. At this point the process enters state Fill A ($y_1 = \top$), and the tank is filled with ingredient A until the tank is filled midway ($x_2 = \top$), at which point the valve for A is closed ($y_1 = \bot$).
3. Next, the system transitions to the state Mix B, in which ingredient B is added (valve B is opened) until $x_3 = \top$. The mixer y_3 is also started in this state.
4. At this point, the system closes the value for B and enters the Drain state ($y_4 = \top$) until the tank is empty (detected by the "low" sensor $x_1 = \bot$). At this point, the mixer is stopped.

The mixing process is an example of a common class of control systems, called sequential control systems. Sequential control systems drive a physical plant through a process consisting of a sequence of discrete steps. Sequential control is used in industrial manufacturing (automotive assembly, QA testing), building automation (elevators, lighting), chemical processing (process control), energy delivery (power management), and transportation automation (railway switching, traffic lights), among others.

Not shown in the example, a *timer* is a special control system primitive that introduces a preset time delay between when an input becomes true, and when a subsequent output becomes true. A timer only sets the specified output to $\top$ when the input has been set to $\top$ for the duration of its preset delay value. Timers can be used to replace other sensors. For example, the level sensors above could be replaced with timers, presuming the flow and drain rates of the apparatus were known and fixed.

2.1 PLC Logic and ISA

The control logic for a sequential process is codified into a set of Boolean circuits that are evaluated in order with dependencies. The circuits are then compiled into the PLC's native instruction set.

The control logic implementation of the chemical mixing process is shown in Figure 1(c). The PLC executes each of the four statements starting with Fill A and ending with Drain once each scan cycle. At the plant start up state, $y_1 = \bot$. After the button x_4 is pressed, $y_1 = \top$ until $x_2 = \top$. Notice that the clause $(x_4 \vee y_1)$ is necessary as x_4 is a button that may only be depressed temporarily, thus once y_1 is activated, it should remain so until the terminating condition $x_2 = \top$. Here, the value of y_1 on the right hand side of the statement is the value for y_1 from the previous scan cycle.

The control output for valve B (y_2) is activated immediately when $y_1 = \bot$ on the condition $x_2 = \top$. y_2 then remains ON until the draining process begins with y_4. The mixer (y_3) is on as long as the level in the tank has reached midway, and has not subsequently been emptied. Finally, the drain (y_4) is activated when the tank is full, and remains ON until the tank is empty.

A logic program must be compiled to a PLC's native instruction set architecture (ISA) before being uploaded to the PLC. While ISAs vary between PLC vendors, many are equivalent to the IEC 61131-3 standard for the Instruction List (IL) programming language [21]. We implement an IL decompiler in Section 3.2.

2.2 Attacking PLCs

Control systems have shown to be vulnerable to attacks through sensors [27], human machine interfaces [2], and PLCs [16, 4]. In this work we focus on the last of these due to the complete control offered to an adversary by PLCs, and the vulnerability of PLCs relative to the other two. For example, hundreds of Internet addressable PLCs can be found on the Shodan computer search engine [1, 12], and new web-enabled, PLCs are being released into the market place with the aims of making remote management more convenient [41, 11]. An even more common arrangement is to have the control system network connected to a corporate network for economic reasons [44, 34]. Given this requirement, it is indeed the case that all but the most critical PLCs will be at least reachable from public networks.

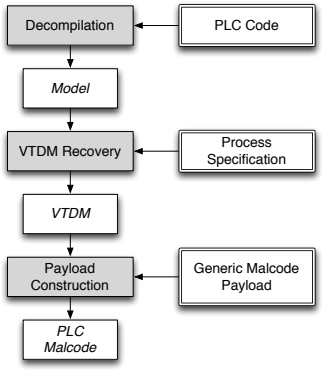


Figure 2: SABOT: Steps to generate a malicious payload.

In this work, we consider an adversary that has sufficient knowledge about the behavior of the target control system to design a targeted and perhaps stealthy payload for that control system. While this may sound like a strong requirement, it is important to note that there are several methods through which one can obtain such information. Many control systems, including railway switching, and electrical substations exhibit some or all of their machinery and behavior in plain view. Furthermore, details about plant structure can be gleaned from vulnerable human machine interfaces [2], and scanning of industrial network protocols [29]. Vendors also release device data sheets and sample control logic, precisely defining device behavior, for example [22]. Of course such information is available to low ranking insiders [24]. Finally, we mention that while it is unlikely that unskilled or "script kiddie" adversaries will mount attacks targeted at specific devices, there is existing work describing such naive attacks and their limitations [29].

3. SABOT

SABOT instantiates malicious payloads for targeted control systems. Depicted in Figure 2, the SABOT initially extracts a logical model of the process from the PLC code (see Section 3.2, Decompilation). Next, the model and process specification are used to create a mapping of physical devices to input and output variables (called the *variable-to-device mapping*), or VTDM, (see Section 3.3, VTDM Recovery). Last, a generic attack is projected onto to the existing model and VTDM to create a malicious payload called the *PLC Malcode* (see Section 3.4, Payload Construction). This malcode is delivered to the victim interface.

3.1 Problem Formulation

Consider a scenario in which an adversary may wish to cause ingredient A to be omitted from the chemical mixing process described above. A PLC payload for this might look like:

$$\text{Valve A} \leftarrow \bot$$
$$\text{Valve B} \leftarrow (\text{Start Button} \vee \text{Valve B}) \wedge \neg\text{Drain Valve}$$

Intuitively, this control system never dispenses A, but rather fills the tank with B. Unfortunately, the adversary does not know how to specify to the PLC which device is meant by "Valve A" or "Start Button." This is because PLCs do not necessarily label their I/O devices with semantically meaningful names like "Drain Valve."[2]

[2]Some PLCs such as the Rockwell Controllogix line allow pro-

Desc.	Bytecode	Accumulator α	Stack
		$\top$	-
And x_1	A x_1	x_1	-
Nested Or	O(	$\top$	$x_1 : \vee$
And x_2	A x_2	x_2	$x_1 : \vee$
And y_1	A y_1	$x_2 \wedge y_1$	$x_1 : \vee$
Pop stack	)	$x_1 \vee (x_2 \wedge y_1)$	-
Store α to y_2	= y_2	$\top$	-
$C \leftarrow C \cup \{y_2 \leftarrow x_1 \vee (x_2 \wedge y_1)\}$			
$V_{\mathcal{M}} \leftarrow V_{\mathcal{M}} \cup \{x_1, x_2, y_1, y_2\}$			

Table 1: Example accumulation of a constraint.

Instead, PLCs use memory addresses, e.g., x_1, y_2, to read values from and write values to sensors and physical devices. We refer to this set of address names as $V_{\mathcal{M}}$. The adversary, who does not know the semantics of the names in $V_{\mathcal{M}}$, prefers to use the set of semantically meaningful names $V_\phi = \{$ Start Button, Valve B, $\ldots\}$.

This raises the problem, *How can an adversary project attack payloads using names in V_ϕ onto a system that uses the unknown memory references $V_{\mathcal{M}}$?* One of SABOT's main tasks is to find a mapping from the names in V_ϕ to those in $V_{\mathcal{M}}$. Here, SABOT requires one additional piece of information from the adversary: a *specification* of the target behavior.

If the adversary is to write a payload such as the one above for the mixing plant, then it is assumed that he knows some facts about the plant. For example, the adversary can make statements like: "The plant contains two ingredient valves, and one drain valve," and "When the start button is pressed, the valve for ingredient A opens." The adversary encodes such statements into a behavioral specification of the target plant. When SABOT is then given a specification and control logic from a plant PLC, it will try to locate the device addresses that behave the same under the rules of the logic as the semantically meaningful names in the adversary's specification.

Like the payload, the sensors and devices specified in the specification are defined using semantically meaningful names from V_ϕ. Given a control logic implementation, SABOT will construct a model $\mathcal{M}$ from the control logic ($\mathsf{Var}(\mathcal{M}) = V_{\mathcal{M}}$), and perform a model checking analysis to find the Variable To Device Mapping (VTDM) $\mu : V_\phi \rightarrow V_{\mathcal{M}}$. SABOT assumes it has the correct mapping μ when all properties in the specification hold under the control logic after their names have been mapped according to μ. For example, the above property, "When the start button is pressed, the valve for ingredient A opens," will be checked as, "Under the rules of the control logic, When x_4 is pressed, then y_1 opens," under the mapping $\mu = \{\text{Start Button} \mapsto x_4, \text{Valve A} \mapsto y_1\}$.

The specification is written as one or more temporal logic formulas ϕ ($\mathsf{Var}(\phi) \subseteq V_\phi$) with some additional hints for SABOT. For a given mapping μ, the adversary supplied payload or specification under μ, denoted $\mu/payload$ or μ/ϕ, is identical to the original, except with any names from V_ϕ replaced by names from $V_{\mathcal{M}}$. Thus, to check whether a given mapping μ maps V_ϕ to the devices is correct, SABOT checks:

$$\mathcal{M} \models \mu/\phi$$

Read, "The temporal logic formula ϕ with literal names mapped by μ holds over the labeled transition system $\mathcal{M}$." If these checks are satisfied under a given μ, then SABOT instantiates the payload over V_ϕ into a payload over $V_{\mathcal{M}}$.

3.2 Decompilation

To obtain a process model $\mathcal{M}$, SABOT must first bridge the gap between the bytecode-level control logic, and the model itself. This means decompiling a list of assembly mnemonics that execute on an accumulator-based architecture into a labeled transition system defined over state variables. SABOT performs this decompilation in two steps. (1) The disassembled control logic bytecode is converted to an intermediate set of constraints C on local, output, and timer variables from the PLC. (2) The constraints in C are then translated to $\mathcal{M}$ using the modeling language of the NuSMV model checker [9].

grammers to give names to I/O ports, but these names are still of no use autonomous malware.

Constraint	NuSMV Model $\mathcal{M}$
input x	VAR x : boolean; ASSIGN init(x) := $\bot$; next(x) := $\{\top, \bot\}$;
output or local y $c = y \leftarrow \alpha$	VAR y : boolean; ASSIGN init(y) := $\bot$; next(y) := α;
timer t $c = t \leftarrow \alpha$	VAR t : boolean, t_p : boolean; ASSIGN init(t) := $\bot$; next(t) := $\alpha \wedge (t_p \vee t)$? $\top$: $\bot$; init(t_p) := $\bot$; next(t_p) := α;

Table 2: Constructing $\mathcal{M}$ from constraints C.

For step 1, the constraints are obtained via symbolic execution of the bytecode. This requires a preprocessing to remove nonstandard instructions not handled by our symbolic execution. The resulting code conforms to the IEC 61131-3 standard for PLC instruction lists [21]. The control flow graph (CFG) of the resulting code is constructed and a symbolic execution is done over the CFG according to a topological ordering. Several register values are tracked, most importantly the logic accumulator α. An example symbolic accumulation of control logic is shown in Table 1.

Step 2 translates the set of constraints resulting from step 1 into a control logic model $\mathcal{M}$ that can be evaluated by the NuSMV model checker. NuSMV takes definitions of labeled transitions systems with states consisting of state variables. SABOT uses the `VAR · : boolean` expression to declare a state variable for each name in $V_{\mathcal{M}}$. Each Boolean variable is first initialized using the `init( · )` expression, and updated at each state transition using the `next( · )` expression. A Boolean variable may be initialized or updated to a constant value of $\top$ or $\bot$, another expression, or a nondeterministic assignment $\{\top, \bot\}$, where both transitions are considered when checking a property. For a complete specification of the NuSMV input language, see [7].

As shown in Table 2, there are three translation rules. In the case of input variables, a new Boolean variable is declared, initialized to $\bot$, and updated nondeterministically. The nondeterministic update is necessary because all possible combinations of sensor readings must be factored into the model. Output and local variables are initialized to $\bot$ and updated according to the expression α.

Timer variables require an extra bit of state. Recall that a PLC timer $t = \top$ only when its input expression $\alpha = \top$ continuously for at least t's preset time duration. Furthermore, any input variable in the model may change state while the timer is expiring. Thus, for each timer, α must hold for two state transitions. The first transition simulates the starting of the timer's countdown, and the second simulates the expiration, allowing the timer to output $\top$.

The decompilation process includes several other steps such as preprocessing bytecode to rewrite vendor specific instructions. Full details can be found in our technical report [30].

3.3 VTDM Recovery

SABOT attempts to find a Variable To Device Mapping (VTDM) μ from names in the adversary's specification to names in the control logic model $\mathcal{M}$. If the correct mapping is found, then the semantics are known for each name in $V_{\mathcal{M}}$ mapped to by μ.

A specification is an ordered list of *properties*. A property with name id has the following syntax:

id : <input [input-list]> <output [output-list]>
<INIT [init-input-list]> <UNIQUE> ϕ

As an example, we can now restate our earlier specification for the plant start button, "When the start button is pressed, the valve for ingredient A opens," as the following property $sbutton$:

$$sbutton \ : \ \texttt{input}\ start^*\ \texttt{output}\ v_{\mathsf{A}}\ \texttt{INIT}\ start^*$$
$$start^* \Rightarrow \mathrm{AX}\ v_{\mathsf{A}}$$

The only mandatory part of a property is the Computational Tree Logic (CTL) formul ϕ [19]. (Interpretations of CTL formulae are given in quotes for the unfamiliar reader.) The CTL formula is defined over names given after the `input` and `output` keywords, where {input-list} $\cup$ {output-list} $\subseteq V_\phi$. SABOT will check ϕ under the control logic model $\mathcal{M}$ in three steps:

1. Choose μ:{input-list}$\cup${output-list}$\rightarrow V_{\mathcal{M}}$.
2. Apply μ to ϕ by substituting all names in ϕ with their mappings in μ. This is denoted by μ/ϕ, read "The property ϕ under the mapping μ."
3. Check $\mathcal{M} \models \mu/\phi$.

These three steps are applied over all possible mappings for a given specification. There are two optional parts to each specification, the list of inputs that will be initially ON `INIT`, and the conflict resolution hint `UNIQUE`. Any names in `init-input-list` will be initialized to $\top$ by the model checker. The keyword `UNIQUE` declares that the names in `input-list` and `output-list` will not appear in any *conflict mappings*.

A conflict mapping is a satisfactory mapping of multiple distinct sets of variables in V_ϕ into a property. Conflict mappings represent ambiguity in the specification, and must be resolved by adding additional properties to the specification. In practice, one or two additional properties are required to resolved such conflicts. The addition of properties only modifies the constant factor of the mapping algorithm's running time. A full discussion of conflict mappings with examples can be found in Section 3.3.2.

3.3.1 Mapping Specifications to Models

SABOT searches for a mapping $\mu : V_\phi \rightarrow V_{\mathcal{M}}$ such that $\mathcal{M} \models \mu/\phi$ for every specification ϕ in the specification. This is done incrementally, finding a satisfying mapping for each specification before moving to the next. If no satisfying mapping is found for a given specification, the previous specification's mapping is discarded, and it is searched again for another satisfying mapping. If no more satisfying mappings are found for the first specification in the specification, the algorithm terminates without identifying a mapping. If a satisfying mapping is found for all specifications, the algorithm accepts this as the correct mapping μ_{SAT} Algorithm 1 shows the basic mapping procedure (except for the `UNIQUE` feature).

We use the NuSMV model checker [9] for deciding $\mathcal{M} \models \mu/\phi$. The running time of IncMapping is dependent on the number of false positive mappings for each specification in the specification bounded below by $\Omega(|specification|)$ and above by $O(|V_{\mathcal{M}}|^{|V_\phi|})$.

Algorithm 1: IncMapping

Input : μ, $spec$, $V_{\mathcal{M}}$, $\mathcal{M}$
Output: The satisfying mapping μ_{SAT} or none

```
1  if spec = ∅ then
2      μ_SAT ← μ
3      return ⊤
4  φ ← Pop(spec)
5  foreach μ_0 : Var(φ) → V_M do
6      if M ⊨ μ_0/φ then
7          if IncMapping(μ ∪ μ_0, spec, V_M − μ_0(Var(φ)), M)
           then
8              return ⊤
9  return ⊥
```

3.3.2 Resolving Conflicts

Consider the problem of writing a specification for the chemical mixer process shown in Figure 1. First, one must define the names in V_ϕ. Denoting input variable names with an '*', let l_1^*, l_2^*, and l_3^* be the names of the low, mid, and high level switches respectively, and let $start^*$ be the start button. Additionally, let v_{A} and v_{B} be the valves for ingredients A and B, $mixer$ be the mixer, and v_d be the drainage valve. As in the figure, $V_{\mathcal{M}} = \{x_1, x_2, x_3, x_4\} \cup \{y_1, y_2, y_3, y_4\}$.

Recall the specification $sbutton$ from above. While $sbutton$ is an accurate specification of plant behavior, it is also ambiguous. To see this, we first consider the case of the correct (true positive) mapping $\mu_{TP} = \{start^* \mapsto x_4, v_{\mathsf{A}} \mapsto y_1\}$. When the mapping is applied to the CTL specification, we get: $\mu_{TP}/sbutton = \texttt{INIT}\ x_4 \ \ x_4 \Rightarrow \mathrm{AX}\ y_1$. μ_{TP} is the correct mapping because, (1) $\mu_{TP}/sbutton$ holds under the control logic (Figure 1(c)), and (2) x_4, and y_1 are the names of the control input and output for the devices that the adversary intended.

Consider an example false positive mapping: $\mu_{FP} = \{start^* \mapsto x_2, v_{\mathsf{A}} \mapsto y_2\}$. Judging by the same criteria as above, we can see that (1) $\mu_{FP}/sbutton$ holds under the control logic, but criterion (2) fails because x_2 and y_2 (the mid level switch and the B ingredient valve respectively) are not the names of control variables intended by the adversary. This raises the question: how can the adversary remove this ambiguity from the specification without *a priori* knowledge of the semantics of x_1, x_2, y_1, and y_2?

While the adversary does not know the semantics of names in $V_{\mathcal{M}}$, he does know the semantics of names in V_ϕ. Thus, the adversary need not know that x_2 is a name for a mid level switch and not a start button, only that there is *some* control variable name that corresponds to a mid level switch. But the adversary already has an abstraction for this, the name l_2^*. The same goes for y_2 and its abstraction, the name v_{B}. Thus, the adversary can reliably remove this ambiguity by checking which names in V_ϕ are in *conflict* with names in the property $sbutton$. For example, consider a specification that has the same structure as $sbutton$ but with different names (Read, "Switch 2 activates valve B"):

$$cflict \ : \ \texttt{input}\ l_2^*\ \texttt{output}\ v_{\mathsf{B}}\ \texttt{INIT}\ l_2^*$$
$$l_2^* \Rightarrow \mathrm{AX}\ v_{\mathsf{B}}$$

Because the correct mapping for $cflict$ differs from the correct mapping for $sbutton$, we have that l_2^* conflicts with $start^*$, and v_{B} conflicts with v_{A}. If the adversary can remove this conflict, then the false positive mapping μ_{FP} will also be removed, and the ambiguity is resolved. The conflict can be removed by the addition of

the following specification (Read, "Valve A can be on with the start button released").

$$sbindep \ : \ \texttt{input}\ start^*\ \texttt{output}\ v_{\mathsf{A}}$$
$$\mathrm{EF}(\neg start^* \wedge v_{\mathsf{A}})$$

To see that the pair $sbutton$, $sbindep$ removes the conflict, we can substitute the conflict into $sbindp$, giving: $\mathrm{EF}(\neg l_2^* \wedge v_{\mathsf{B}})$, which does not hold under the control logic. Thus, if the conflicting mapping is initially made when checking $sbutton$, this mapping will fail when checking $sbindep$, which will cause the SABOT checker to go back to $sbutton$, and search for another mapping, in this case, the correct one. Of course, this approach is not guaranteed to make a completely unambiguous specification (as will be seen in Section 4), but it does remove all ambiguities with respect to devices the adversary is aware of.

In larger examples, we keep track of conflicts and resolutions using *conflict tables*. An example set of conflict tables for a specification of a chemical process is shown in Figure 3. Each specification has a (potentially empty) conflict table listing all unmapped names in V_ϕ that satisfy the specification. Given a nonempty conflict table for a specification $spec$, one can make $spec$ unambiguous by writing at most one additional specification over the names in $spec$ for each entry in its conflict table. This guarantees a finite specification size bounded in $O(|V_\phi|^2)$ specifications. Significantly fewer are usually necessary in practice.

The specification keyword `UNIQUE` is used to declare that no name appearing in the specification will appear in a conflicting mapping. This is useful because there are some cases where the same name appears in many conflict table entries. (See specification $sbutton$ in Figure 3). It is never required to use `UNIQUE` to remove conflicts, but it can be useful in reducing the search space.

3.4 Payload Construction

In the payload construction step, SABOT instantiates a generic malicious payload for the target PLC. Given a VTDM μ for a victim PLC and specification, SABOT uses μ to map the names in the adversary's generic payload into those in the control logic. Thereafter, the instantiated payload is recompiled into bytecode and uploaded to the PLC.

SABOT payloads are control logic programs defined over names in V_ϕ. Once the VTDM μ is found, the payload is *instantiated* under μ, producing a payload over names in $V_\mathcal{M}$. Using this approach, an adversary can assume the same semantics for names in the payload that are assumed for names in the specification. As an example, the following payload manipulates the chemical mixing process into omitting ingredient A from the mixture.

Generic Payload	**Instantiated Payload**
$v_{\mathsf{A}} \leftarrow \bot$	$y_1 \leftarrow \bot$
$v_{\mathsf{B}} \leftarrow (start^* \vee v_{\mathsf{B}}) \wedge \neg l_3^*$	$y_2 \leftarrow (x_4 \vee y_2) \wedge \neg x_3$
$mixer \leftarrow (l_2^* \vee mixer) \wedge l_1^*$	$y_3 \leftarrow (x_2 \vee y_3) \wedge x_1$
$v_d \leftarrow (l_3^* \vee v_d) \wedge l_1^*$	$y_4 \leftarrow (x_3 \vee y_4) \wedge x_1$

Assuming that the correct mapping, i.e. μ_{TP} is found by SABOT, then this payload will execute as expected when uploaded to the mixer PLC. Our results in Section 4.1 show that the correct mapping can be recovered the majority of the time, even when the plant has unexpected devices and functionality. If the adversary is unaware of a device in the plant, then even a correct VTDM may cause side effects when used to instantiate an attack. The cause of such side effects can be concealed by malicious code execution on the PLC programming machine [16].

System	Baseline	Emergency	Annunciator	Sequential	Parallel
Container Filling					
Motor Control		■			
Traffic Signaling			■		
pH Neutralization					
Railway Switching			■		

Table 3: Control system variants (omitted shaded).

4. EVALUATION

We evaluate SABOT using four metrics:

- **Accuracy** is the ability of SABOT to correctly map a specification onto a control system, even when unexpected features are present. We evaluate the number of devices that are correctly mapped in each test case in Section 4.1.
- **Adaptability** is the ability to recognize different variants of a control system. A canonical specification is tested against two unique implementations of a traffic light control logic, and results evaluated in Section 4.2.
- **Performance** is the ability to efficiently recover the VTDM for a given specification. We evaluate not only the total running time, but the number of false positives encountered by the incremental mapping process in Section 4.3.
- **Scalability** is the ability to perform accurately and efficiently as additional control system functionality is added to the PLC. We augment SABOT with a basic dependency graph analysis, and evaluate both accuracy and performance for PLCs running multiple independent subsystems Section 4.4.

Note that this is the first set of experiments on SABOT, and additional future studies providing further validation and enlightenment are appropriate. However, the results of this initial study indicate that SABOT works well on a variety of control logics, and that there remain numerous avenues for engineering and fine tuning results. We evaluate the use of two such improvements to refine our initial results in section 4.5.

All experiments are conducted using the five variants of five representative process control systems taken from real-world applications described in Table 3. Each process description is used to implement a specification and a control logic. All specifications were created independently of the implementations. Note that control variables are denoted using *emphasis* and input variable are annotated with '*' (e.g., $invar^*$, $outvar$) below.

While seemingly simple, most of the following systems have a larger variety of devices and more complex state machines than the target of the Stuxnet attack, many uniform variable speed drives [16]. The five applications are:

Container Filling. Consider the filling of product containers on an assembly line, e.g., cereal boxes [13]. In the basic process, a belt $belt$ carries an empty container ($belt = \top$) until it is under the fill-bin as indicated by a condition $cond^*$, e.g., a light barrier detects when container is in position. Thereafter, a fill valve $fill$ is opened for a period of time to fill, and the belt is activated to move the next container into place. It may also occur that the fill bin itself

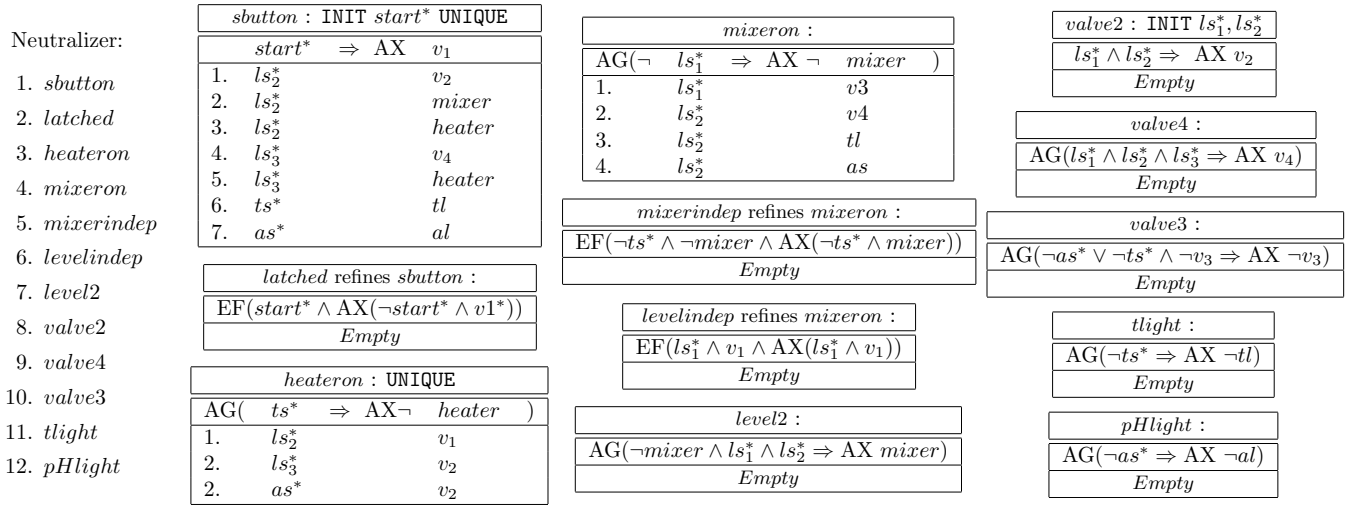


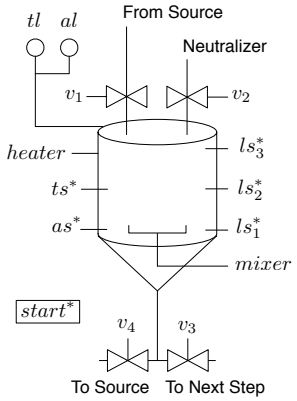
Figure 3: An unambiguous specification with conflict tables for the pH neutralization system.

is depleted, as indicated by a level sensor low^* and a secondary source bin with valve $source$ will be used to replenished the fill bin.

Motor Control. Stepper motors divide a full rotation into a set of discrete angular positions. This has many applications in precision equipment like lathes, conveyors, and applications requiring bidirectional operation. A stepper motor controller allows for starting the motor in the forward direction fon and reversing the motor ron, such that $\neg fon \vee \neg ron$ always holds. Buttons are used for selecting forward (for^*) and reverse (rev^*) operation, and stopping the motor $stop^*$. The controller will also enforce a minimum spin down time before the motor is allowed to change direction.

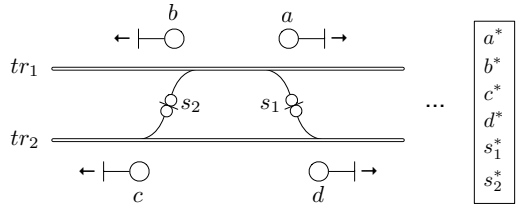
Traffic Signaling.[3] Consider further a typical 4-way traffic light with red_1, $yellow_1$, and $green_1$ in the North/South direction, and red_2, $yellow_2$, and $green_2$ in the East/West direction. Ignoring inputs from pedestrians or road sensors, the traffic light follows the cycle: (red_1, red_2), $(red_1, green_2)$, $(red_1, yellow_2)$, (red_1, red_2), $(green_1, red_2)$, $(yellow_1, red_2)$.

pH Neutralization. A pH neutralization system mixes a substance of unknown acidity with a neutralizer until the desired pH level is achieved, e.g., in wastewater treatment. An example process adapted from [15] operates as follows:



When the tank level is below ls_2^*, valve v_1 is opened to fill the tank with the acidic substance up to level ls_2^*. At this point valve v_2 is opened to dispense the neutralizer until the correct acidity level is indicated by the acidity sensor $as^* = \top$ and the correct temperature of the product has been reached as indicated by temperature sensor $ts^* = \top$. If $ts^* = \bot$, the $heater$ is activated. If the desired pH level is not achieved before the tank fills to ls_3^*, v_2 is closed and v_4 is opened to drain the tank back to level ls_2^*. Once the correct pH level and temperature are achieved, and there is at least ls_2^* liquid in the tank, it is drained to the next stage by v_3. The temperature and acidity lights tl and al are activated when the desired temperature and acidity have been reached respectively, and the tank is at least at level ls_2^*.

Railway Switching. Lastly, consider a process that safely coordinates track switches and signal lights on a real railway segment [17]:



The segment consists of the two tracks (tr_1 and tr_2), two switches (s_1 and s_2) and four signal lights a–d. A switch is said to be in the *normal* state if it does not allow a train to switch tracks and in the *reverse* state if it does allow a train to switch tracks. If a signal is ON, it indicates that the train may proceed and if it is OFF, the train must stop. The direction of each signal is indicated by an arrow.

The signalman can control the state of the lights and switches using the inputs s_1^*, s_2^* and a^*–d^*, where $s_1^* = \top$ puts switch s_1 into reverse, and a^* turns signal a ON. To ensure safe operation, the control logic maintains several invariants over the signal and switching commands from the signalman. (1) Two signals on the same track, e.g., a and b, can never be ON simultaneously. (2) If a switch is in reverse, then at most one signal is allowed to be ON. (3.a) If switch s_1 is in reverse, then signals a and c must be OFF. (3.b) If switch s_2 is in reverse, then signals b and d must be OFF. (4) If both switches are set to reverse, then all signals must be OFF.

[3]Traffic signal attacks can cause significant congestion [18].

Control Logic	Container Filling	Start Button	Belt	Fill Valve	Synchronized	Motor Control	Forward Button	Reverse Button	Stop Button	Motor Forward	Motor Reverse	Synchronized	Traffic Signal	Red Light 1	Red Light 2	Green Light 1	Green Light 2	Yellow Light 1	Yellow Light 2	Synchronized
Baseline																				
Emergency														n	n	n	n	n	n	
Annunciator																				
Sequential																				
Parallel																				

Control Logic	pH Neutralization	Start Button	Source Valve	Neutralizer Valve	To Source Valve	Product Valve	Heater	Mixer	Temp Sensor	Acidity Sensor	Low Level Switch	Mid Level Switch	High Level Switch	Synchronized	Railway Switching	Signal A	Signal B	Signal C	Signal D	Switch 1	Switch 2	Synchronized
Baseline																						
Emergency	n	n	n	n	n	n	n	n	n	n	n	n										
Annunciator			p																			
Sequential			p	p	p								p									
Parallel			p	p	p	p		p	p			p	p									

Table 4: Per-device accuracy results. Empty cell: Correct mapping, 'p': false positive mapping, 'n':false negative (no mapping), Shaded cell: experiment omitted (see description).

Each of the applications is implemented in a **Baseline** control system. Each baseline system is then modified with four variants to introduce plant features not anticipated by the baseline specifications:

Emergency. This case adds an emergency shutdown button named $estop^*$ to each plant. If the emergency shutdown button is pushed, all devices will be immediately turned OFF. One can see how this can cause false negative mappings. For example, a property of the form $\mathrm{AG}(input^* \Rightarrow \mathrm{AX}\ output)$ will no longer hold, because of the case where $input^* \wedge estop^*$ holds, but $output$ is forced to $\perp$. The motor controller's stop button acts as a shutdown, so there was no need to add one.

Annunciator. *Annunciator panels* are visual or sometimes audible displays present within the plant itself. We place a single annunciator light on each input and output in the plant. This light is turned ON by the control logic if the corresponding input or output is ON, and OFF otherwise. We evaluate the plants with annunciator panels as they nearly double the number of control variables that are expected by the adversary. The traffic signal and railway switching processes were not evaluated with annunciator panels, as annunciator functionality is already present in both systems.

Sequential. This case considers a plant with two distinct instances of the process, where the second instance is dependent on the first. For the motor controller, this simply means that the second motor mimics the forward and reverse behavior of the second. The same is true for the traffic lights, where the state of the second mimics the first. For sequential container filling, containers are partially by the first system, then by the second. The railway switching example is modified to include three tracks, and allow a train to switch from the first track, to a middle second track, and then to the third track. The safety properties are extended to prevent any conflicting routes between the three tracks. Finally, the chemical neutralization process is serialized to two tanks, such that the first process fully drains to the second before the second starts.

Parallel. This case models two independent instances of the process executing in parallel on the same PLC. This is expected to occur in production environments where it is more cost effective to add more input and output wires to the same PLC than to maintain distinct PLCs for each parallel instance of the process. A special criterion is added for the parallel case called **Synchronized**, which is true if all mappings where true positives in the same instance of the process. I.e., there is no mixing of mappings between the two independent processes.

4.1 Accuracy

Recall that accuracy is a measure of the correctness of the identified mappings between internal devices and the process specifications, i.e., the accuracy of the VTDM. Here we measure correctly mapped devices, incorrect mappings (false positives), and failures to identify any mapping at all (false negatives). The results for accuracy experiments are shown in Table 4. To summarize, in three out of five test systems the baseline specification is sufficient to produce a complete, correct mapping for the control system, and four out of five systems had no false positives.

As expected, the emergency shutdown case caused false negatives in two out of the three control systems. This was due to the failure of specifications of the form $\mathrm{AG}(input^* \Rightarrow output)$ to always hold under any mapping when there is always a state in which $estop^*$ makes the property not hold. The false negatives occur for all devices in both cases, because later specifications contained names that failed to map in earlier specifications, making them uncheckable.

The pH neutralization system experienced the most false positives due to its multiple parallel behaviors. In the annunciator panel, the behavior of the valve for the neutralizer (v_2) could not be distinguished from the annunciator light for the mid level switch ls_2^* because $ls_2^* = \top$ implies that both the valve and annunciator light are ON. More broadly, the sequential system false positives were caused by devices in the first instance that were mistaken for a device in the second instance. For example, the heater in the

System	Baseline	Emergency	Annunciator	Sequential	Parallel
Container Filling	0.59/16	0.68/21	1.03/28	1.00/27	1.57/29
Motor Control	0.23/7	-/-	0.26/7	0.45/14	0.86/19
Traffic Signal	5.59/125	6.35/130	-/-	51.76/1040	103.60/1385
pH Neutralization	2.42/70	3.67/101	4.55/100	14.16/228	11.51/179
Railway Switching	0.92/25	1.05/27	-/-	4.00/97	19.95/97

Table 5: Running time (s) / number of calls to the model checker for each system and case.

first instance of the pH neutralization system was confused with the heater in the second. The false positives in the parallel case were caused by devices in the first instance that were mistaken for non-equivalent devices in the second. For example, the finished product valve (v_3) in the first instance of the pH neutralization system was confused with the mixer in the second instance. We improve upon these results in Section 4.5.

4.2 Adaptability

Adaptability is the ability to recognize a control system by its behavior, independent of its implementation. Because SABOT only considers control logic *behavior* and not its *structure*, any implementation conforming to the process description will be handled equivalently. To confirm this is the case, and as an experimental control, a team member not involved in prior analysis was tasked with implementing an alternative traffic signal control program that exhibited the behavior from the description. The team member took an alternative strategy of allowing the light timers to drive the rest of the process, resulting in a significantly different implementation. The same experiments run above were rerun on this new implementation, and the results were identical.

4.3 Performance

To gauge runtime costs, we measured the running time and number of calls to the NuSMV model checker of each experiment conducted in Section 4.1. Note that SABOT's running time for a given model and specification is hard to calculate for a given set of inputs because it is highly dependent on the number of incremental mappings attempted.

Shown in Table 5, in over 75% of tests, the mapping is found in less than 10 seconds, and in 90% of tests, the mapping is found in less than 30 seconds. Two tests however deserve particular attention. The test with a running time of 1m43.6s for parallel traffic signaling made the most calls to the model checker of any test. It also represented the greatest increase in calls to the model checker over its own baseline. This can be attributed to the fact that the traffic signal was the only specification containing specifications that mapped three names at once.

Also note the comparative running times and number of checker calls for the sequential and parallel railway switching tests. Both required the same number of calls to the checker, but the parallel case has nearly a fivefold increase in running time. The extended running time for each call to the checker was the result of the difference in state space between the two. In the sequential case, there were two systems with one set of inputs, and the second system dependent on the first. In the parallel case, the independent input sets greatly inflated the model's state space. We discuss a check that could reduce state space explosion in the following section.

4.4 Scalability

Thus far, we have assumed that the control logic only contained the specified functionality. In this section, we evaluate SABOT's accuracy and performance given a PLC that has functionality for additional independent subsystems in place. In practice, a large facility such as a nuclear power plant or waste water treatment facility will be broken down into subsystems, each of which will have a dedicated PLC. These PLCs are in turn coordinated by higher levels of *supervisory control*. However, this does not guarantee that additional unspecified functionality will not be run on the same PLC. Thus, we wish to evaluate SABOT under such a scenario.

Specified	Run time (s)	Calls	FP
Container Filling	3.28	96	0
Motor Control	4.70	138	1
Traffic Signal	18.28	485	0
pH Neutralization	6.49	174	0
Railway Switching	75.70	2057	0

Table 6: Run time (s), checker calls, and false positives for checks against the monolithic control logic.

In this section, we augment SABOT with a simple dependency analysis that separates the control logic into separate models for each independent subsystem. The subsystem models are constructed as follows. First, the variable dependency graph for the control logic is constructed. Second, an *undirected* edge is added between any two output variables with a common dependency. Third, a new graph is constructed using just the output variables and newly added undirected edges. A single subsystem model is constructed for each strongly connected component in this graph. The specification is then tested against each model independently.

To simulate independent subsystems running in the same PLC, we combined all five test systems into a single monolithic control logic. We then ran SABOT with the dependency analysis against the monolithic logic with each of the five specifications. Ideally, in each run, SABOT would match the specification only to the correct corresponding subsystem. There are two types of errors that can occur here. First, a specification could be mapped to an incorrect subsystem. Second, an incorrect dependency analysis may occur, e.g., if variables in multiple subsystems share a dependency. While this was not an issue in our experiments, we defer a more sophisticated dependency analysis algorithm to future work.

The performance and accuracy results are shown in Table 6. Only a single false positive mapping occurred from the specification for motor control onto the implementation of railway switching. Unlike the performance results in Section 4.3, here, the railway switching has the highest running time. This is due to the large number of incremental false positive mappings that occur when testing the railway switching specification against pH neutralization, which led to many false positive variable mappings being rejected by the incremental mapping algorithm. Nevertheless, in all cases, the run times are still within the limits found in Section 4.3.

4.5 Accuracy Improvements

In this section, we consider using two refinements to improve the accuracy results found in Section 4.1. First, we include the dependency analysis introduced in the previous section in hopes of improving the results for the parallel variation in pH neutralization. Second, we introduce a method to *safeguard* a specification against the presence of emergency stop systems, a common feature.

Control Logic	pH Neutralization / Start Button	Source Valve	Neutralizer Valve	To Source Valve	Product Valve	Heater	Mixer	Temp Sensor	Acidity Sensor	Low Level Switch	Mid Level Switch	High Level Switch	Synchronized	Traffic Signal / Red Light 1	Red Light 2	Green Light 1	Green Light 2	Yellow Light 1	Yellow Light 2	Synchronized
Baseline																				
Emergency																				
Annunciator			p																	
Sequential			p	p	p							p								
Parallel																				

Table 7: Accuracy improvements on results from Table 4.

We safeguard a specification against an emergency stop button by slightly weakening each property of the form $\mathrm{AG}(\psi \Rightarrow \phi)$ to allow it to fail on cases where $\psi \wedge estop$ holds. For example, in the pH neutralizer, the property $\mathrm{AG}(\neg mixer \wedge ls_1^* \wedge ls_2^* \Rightarrow \mathrm{AX}\, mixer)$ was changed to $\mathrm{AF}(\neg mixer \wedge ls_1^* \wedge ls_2^* \Rightarrow \mathrm{AX}\, mixer)$, and a fairness constraint `FAIRNESS` $mixer$ was added to force the model checker to ignore the path on which $estop$ was infinitely ON. For properties of other forms, such as $\mathrm{AG}(\neg\psi \Rightarrow \phi)$, no such weakening is necessary because it still holds when either $\psi \wedge estop$ or $\psi \wedge \neg estop$.

The results with both dependency analysis and safeguards are shown in Table 7. The safeguard was applied to all properties in the pH neutralization and traffic signal systems, and all five test cases were rerun. The addition of the safeguards did not negatively affect accuracy in the baseline case, and the number of test cases for which all devices are mapped regardless of plant features is increased to 4 out of 5. Thus, if an adversary has reason to suspect that an emergency stop system may be in place, the use of safeguards can be an effective workaround.

5. COUNTERMEASURES

We now explore several avenues to countermeasures to SABOT-like mechanisms in control systems: *improved perimeter security*, *novel PLC security architectures*, and *control logic obfuscation.*

Improved perimeter security. Perhaps the most straightforward way to safeguard against malicious PLC payloads is to improve perimeter defenses around PLCs. Unfortunately, the most effective solution, disconnecting the PLC from any networked computer, is neither common practice, nor in many cases economically feasible. Due to economic constraints and for ease of maintenance, PLCs are often connected to the corporate networks [44] or the Internet [41, 11]. However numerous standards exists for defense in depth security in control systems across industries [14] such as in the power sector [42, 32]. Compliance with standards, however, can lead to a checklist approach to security that can ultimately give a false sense of security impact [43, 34]. A final defense-in-depth technique is the use of SCADA honeynets [36] outside of the protected perimeter to detect adversaries before they access production SCADA devices.

Novel PLC security architectures. PLCs as they exist today support virtually no security precautions short of basic passwords. There are several basic architectural changes that can mitigate any PLC payload mechanism. Mohan et al. [31] proposed the use of a safety PLC to monitor plant behavior and detect deviations from deterministic behavior. Similarly, a model based anomaly detection system for SCADA networks was proposed in [8]. Lemay et al. [25] used attestation protocols to verify the integrity the code running on a smart electricity meter, including firmware updates. Attestations have also been used to prevent peripheral firmware from attacking a host computer [26]. A similar method could be applied to control systems in which PLCs must attest their entire stack, including firmware and control logic, to a trusted third party before being allowed to send control signals to devices. Thus, any maliciously uploaded control logic would be discovered. Of course, any such solution faces a long path to deployment in real world systems.

Control logic obfuscation. If the above two measures fail, obfuscation of the PLC's control logic can offer a final line of defense in preventing SABOT's analysis. This has the added advantage that no modification of the control system is required beyond obfuscating the control logic itself. Much of the existing work on program obfuscation is has attempted either to evade malware signature matching [39] or to prevent code injection into address spaces [40]. Here, the objectives are different; the goal in this case is to prevent either decompilation or VTDM recovery by SABOT. Attempts at preventing decompilation will likely fail, as control engineers expect to be able to read code from a PLC and decompile it. (Most PLC development environments can decompile control logic.) A more promising route is to add noise in the form of unused variables that are not mapped to any devices. This would, however, be a fundamental departure from current control engineering practices.

6. RELATED WORK

There are many tools preceding SABOT aimed at exploiting software vulnerabilities in control systems. Automated exploit frameworks like Metasploit [28] have been extended to attacks against SCADA and control systems [23]. While such exploitation of control systems is however not new [5], the rate of release of new SCADA and PLC exploits has been accelerating [35, 10]. Such tools can be seen as fulfilling the first step of control system penetration, the subsequent step being automatic payload generation against the victim PLC.

McLaughlin first described the requirements for extending automated exploit frameworks to perform attacks against PLCs in [29]. The focus is, however, how to execute untargeted attacks against an PLC system based on techniques such as the violation of safety interlocks found in the control logic. The topic of executing a targeted attack is only briefly touched on, as it is mentioned that a means will be necessary for an adversary to specify the *payload goal*. However, no means is given for actually specifying the goal, or for achieving it. SABOT is, to the best of our knowledge, the first evaluation of these techniques.

7. CONCLUSION

In this paper, we have presented SABOT as a means to lower the bar of sophistication needed to construct payloads for vulnerable PLCs. If an adversary is familiar enough with their target to specify a precise attack definition, then SABOT can map a supplied behavioral specification onto the code from the victim PLC, allowing a malicious payload to be instantiated. We have demonstrated that even when unexpected features or independent subsystems are implemented in a PLC, SABOT can still find a sufficient mapping to instantiate a payload for the system within a reasonable time frame. PLC code obfuscation may prove an effective defense against such attacks, though it is in conflict with current practices.

8. REFERENCES

[1] Shodan. http://www.shodanhq.net.

[2] ADVANTECH/BROADWIN WEBACCESS RPC VULNERABILITY. ICS-CERT Advisory 11-094-02, April 2011.

[3] AMIN, S., LITRICO, X., SASTRY, S. S., AND BAYEN, A. M. Stealthy Deception Attacks on Water SCADA Systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control* (2010).

[4] BERESFORD, D. Exploiting Siemens Simatic S7 PLCs. In *Black Hat USA* (2011).

[5] BYRES, E., AND LOWE, J. The Myths and Facts behind Cyber Security Risks for Industrial Control Systems. In *ISA Process Control Conference* (2003).

[6] CÁRDENAS, A. A., AMIN, S., AND SASTRY, S. Research Challenges for the Security of Control Systems. In *Proceedings of the 3rd conference on Hot topics in security* (2008).

[7] CAVADA, R., CIMATTI, A., JOCHIM, C. A., KEIGHREN, G., OLIVETTI, E., PISTORE, M., ROVERI, M., AND TCHALTSEV, A. NuSMV 2.5 User Manual, 2010.

[8] CHEUNG, S., DUTERTRE, B., FONG, M., LINDQVIST, U., SKINNER, K., AND VALDES, A. Using Model-based Intrusion Detection for SCADA Networks. In *Proceedings of the SCADA Security Scientific Symposium* (2007).

[9] CIMATTI, A., CLARKE, E., GIUNCHIGLIA, F., AND ROVERI, M. NuSMV: A New Symbolic Model Verifier. In *Computer Aided Verification*. Springer Berlin / Heidelberg, 1999.

[10] CONSTANTIN, L. Researchers Expose Flaws in Popular Industrial Control Systems. http://www.pcworld.com, January 2012.

[11] CONTROL TECHNOLOGY CORP. Blue Fusion: Model 5200 Controller. http://www.ctc-control.com/products/5200/5200.asp.

[12] ÉIREANN P. LEVERTT. Quantitatively Assessing and Visualising Industrial System Attack Surfaces. Master's thesis, University of Cambridge, 2011.

[13] ERICKSON, K. T., AND HEDRICK, J. L. *Plantwide Process Control*. Wiley Inter-Science, 1999.

[14] EVANS, R. P. Control Systems Cyber Security Standards Support Activities, January 2009.

[15] FALCIONE, A., AND KROGH, B. Design Recovery for Relay Ladder Logic. In *First IEEE Conference on Control Applications* (1992).

[16] FALLIERE, N., MURCHU, L. O., AND CHIEN, E. W32.Stuxnet Dossier. http://www.symantec.com, 2010.

[17] FERREIRA, N. G., AND SILVA, P. S. M. Automatic Verification of Safety Rules for a Subway Control Software. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF)* (2004).

[18] GRAD, S. Engineers who hacked into L.A. traffic signal computer, jamming streets, sentenced. http://latimesblogs.latimes.com.

[19] HUTH, M., AND RYAN, M. *Logic in Computer Science*. Cambridge University Press, 2004.

[20] IGURE, V. M., LAUGHTER, S. A., AND WILLIAMS, R. D. Security Issues in SCADA Networks. *Computers and Security* (2006).

[21] INTERNATIONAL ELECTROTECHNICAL COMMISSION. International Standard IEC 61131 Part 3: Programming Languages.

[22] KEYENCE AMERICA. Position control using a stepper motor. http://www.keyence.com/downloads/plc_dwn.php.

[23] LANGILL, J. White Phosphorus Exploit Pack Ver 1.11 Released for Immunity Canvas. http://scadahacker.blogspot.com, 2011.

[24] LEALL, N. Lessons from an Insider Attack on SCADA Systems. http://blogs.cisco.com/security/lessons_from_an_insider_attack_on_scada_systems, August 2009.

[25] LEMAY, M., AND GUNTER, C. A. Cumulative attestation kernels for embedded systems. In *Proceedings of the 14th European Symposium on Research in Computer Security (ESORICS)* (2009).

[26] LI, Y., MCCUNE, J. M., AND PERRIG, A. Viper: verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*.

[27] LIU, Y., NING, P., AND REITER, M. K. False Data Injection Attacks against State Estimation in Electric Power Grids. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (November 2009).

[28] MAYOR, D., MOOKHEY, K. K., CERVINI, J., AND ROSLAN, F. *Metasploit Tookit: for Penetration Testing, Exploit Devevlopment, and Vulnerability Research*. Syngress, 2007.

[29] MCLAUGHLIN, S. On Dynamic Malware Payloads Aimed at Programmable Logic Controllers. In *6th USENIX Workshop on Hot Topics in Security* (2011).

[30] MCLAUGHLIN, S., AND MCDANIEL, P. SABOT: Specification-based Payload Generation for Programmable Logic Controllers. Tech. Rep. NAS-TR-0162-2012, The Pennsylvania State University, 2012.

[31] MOHAN, S., BAK, S., BETTI, E., YUN, H., SHA, L., AND CACCAMO, M. S3A: Secure System Simplex Architecture for Enhanced Security of Cyber-Physical Systems. http://arxiv.org, 2012.

[32] NATIONAL ENERGY REGULATORY COMISSION. NERC CIP 002 1 - Critical Cyber Asset Identification, 2006.

[33] NICOL, D. M. Hacking the Lights Out. *Scientific American 305*, 1 (July 2011), 70–75.

[34] PIÈTRE-CAMBACÉDÈS, L., TRISCHLER, M., AND ERICSSON, G. N. Cybersecurity Myths on Power Control Systems: 21 Misconceptions and False Beliefs. *IEEE Transactions on Power Delivery* (2011).

[35] POLLET, J. Electricity for free? the dirty underbelly of scada and smart meters. In *Proceedings of Black Hat USA 2010* (July 2010).

[36] POTHAMSETTY, V., AND FRANZ, M. The SCADA Honeynet Project. http://scadahoneynet.sourceforge.net.

[37] ROBERTS, P. F. Zotob, PnP Worms Slam 13 DaimlerChrysler Plants. http://www.eweek.com, August 2008.

[38] SANGER, D. E. Obama Order Sped Up Wave of Cyberattacks Against Iran. *New York Times* (June 2012).

[39] SZOR, P. *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005.

[40] THE PAX TEAM. Address Space Layout Randomization of PaX. pax.grsecurity.net.

[41] TRIANGLE RESEARCH INTERNATIONAL. Connecting Super PLCs to the Internet. http://www.tri-plc.com/internetconnect.htm.

[42] U.S. DEPARTMENT OF ENERGY OFFICE OF ELECTRICITY DELIVERY AND ENERGY RELIABILITY. A Summary of Control System Security Standards Activities in the Energy Sector, October 2005.

[43] WEISS, J. Are the NERC CIPS making the grid less reliable. *Control Global* (2009).

[44] YARDLEY, T. SCADA: Issues, Vulnerabilities, and Future Directions. *;login 34*, 6 (December 2008), 14–20.