Orpheus: Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks

Long Cheng
Department of Computer Science
Virginia Tech, USA
chengl@vt.edu

Ke Tian
Department of Computer Science
Virginia Tech, USA
ketian@vt.edu

Danfeng (Daphne) Yao
Department of Computer Science
Virginia Tech, USA
danfeng@vt.edu

ABSTRACT

Recent studies have revealed that control programs running on embedded devices suffer from both control-oriented attacks (*e.g.*, code-injection or code-reuse attacks) and data-oriented attacks (*e.g.*, non-control data attacks). Unfortunately, existing detection mechanisms are insufficient to detect runtime data-oriented exploits, due to the lack of runtime execution semantics checking. In this work, we propose *Orpheus*, a security methodology for defending against data-oriented attacks by enf<u>or</u>cing cyber-<u>ph</u>ysical ex<u>e</u>c<u>u</u>tion <u>s</u>emantics. We address several challenges in reasoning cyber-physical execution semantics of a control program, including the event identification and dependence analysis. As an instantiation of *Orpheus*, we present a new program behavior model, *i.e.*, the event-aware finite-state automaton (*e*FSA). *e*FSA takes advantage of the event-driven nature of control programs and incorporates event checking in anomaly detection. It detects data-oriented exploits if physical events and *e*FSA's state transitions are inconsistent. We evaluate our prototype's performance by conducting case studies under data-oriented attacks. Results show that *e*FSA can successfully detect different runtime attacks. Our prototype on Raspberry Pi incurs a low overhead, taking 0.0001s for each state transition integrity checking, and 0.063s~0.211s for the cyber-physical contextual consistency checking.

CCS CONCEPTS

• **Security and privacy → Software and application security**; **Intrusion/anomaly detection and malware mitigation**; • **Computer systems organization → Embedded and cyber-physical systems**;

KEYWORDS

Anomaly detection; Cyber-physical systems; Data-oriented attacks; Control programs; Execution semantics; Event awareness;

1 INTRODUCTION

Embedded control and monitoring systems are becoming widely used in a variety of CPS (Cyber-Physical Systems) and IoT (Internet of Things) applications[1], including building and home automation, medical instrumentation, automotive, as well as operations of critical infrastructures such as chemical and nuclear facilities, electric power grid, oil and natural gas distribution [51]. In these applications, embedded systems software monitors physical environments by taking sensory data as input and makes control decisions that affect physical environments or processes [40]. We refer to such embedded systems software in CPS/IoT applications as cyber-physical control programs, *a.k.a.* control programs.

Control programs are critical to the proper operations of CPS/IoT, as anomalous program behaviors can have serious consequence, or even cause devastating damages to physical systems [11]. For example, the Stuxnet [33] malware allowed hackers to compromise the control system of a nuclear power plant and manipulate physical equipment such as centrifuge rotor speeds. Therefore, protecting control programs against malicious attacks becomes of paramount importance in cyber-physical applications. Recent studies [11, 18, 27, 29, 42, 65] have shown that control programs suffer from a variety of runtime software exploits. These attacks can be broadly classified into two categories:

- *Control-oriented attacks* exploit memory corruption vulnerabilities to divert a program's control flows, *e.g.*, malicious code injection [24] or code reuse attacks [29]. Control-oriented attacks in conventional cyber systems (*i.e.*, without cyber-physical interactions) have been well studied [50]. It is possible that existing detection approaches [9, 10, 17, 25, 36] are extended to defend against control-oriented attacks in embedded systems software.
- *Data-oriented attacks*[2] manipulate program's internal data variables without violating its control-flow integrity (CFI), *e.g.*, non-control data attacks [19], control-flow bending [17], data-oriented programming [31]. Data-oriented attacks are much more stealthy than attacks against control flows. Because existing CFI-based solutions are rendered defenseless under data-oriented attacks, such threats are particularly alarming.

Since many control decisions are made based on particular values of data variables in control programs [11], data-oriented attacks could potentially cause serious harm to physical systems in a stealthy way. We further categorize data-oriented attacks against control programs into two types. *i)* ***Attacks on control branch***, which corrupt critical decision making variables at runtime to execute a *valid-yet-unexpected* control-flow path (*e.g.*, allowing liquid

ACSAC 2017, December 4–8, 2017, San Juan, PR, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134640

[1]Though CPS and IoT are defined with different emphasis and have no standard definitions agreed upon by the research community, they have significant overlaps. In general, CPS emphasizes the tightly coupled integration of computational components and physical world. While IoT has an emphasis on the connection of things with networks. If an IoT system interacts with the physical world via sensors/actuators, we can also classify it as a CPS [2].

[2]We mainly focus on runtime software exploits, and thus sensor data spoofing attacks in the physical domain are out of the scope in this work.

to flow into a tank despite it is full [12] or preventing a blast furnace from being shut down properly as in the recent German steel mill attack [3]). *ii)* ***Attacks on control intensity***, which corrupt sensor data variables to manipulate the amount of control operations, *e.g.*, affecting the number of loop iterations to dispense too much drug [11]). These data-oriented attacks result in inconsistencies between the physical context and program execution, where executed control-flow paths do not correspond to the observations in the physical environment. Unfortunately, there exist very few defenses [11, 62] and they are ineffective to prevent both attack types due to the lack of runtime execution semantics checking.

In many instances, a CPS/IoT system can be modeled as an *event-driven* control system [22, 32]. We refer to events as occurrences of interest that come through the cyber-physical observation process or emitted by other entities (*e.g.*, the remote controller), and trigger the execution of corresponding control actions. In this paper, we point out the need for an event-aware control-program anomaly detection, which *reasons about program behaviors with respect to cyber-physical interactions*, *e.g.*, whether or not to open a valve is based on the current ground truth water level of a tank [12]. None of existing program anomaly detection solutions [50] has the event-aware detection ability. They cannot detect attacks that cause inconsistencies between program control flow paths and the physical environments.

We address the problem of securing control programs against data-oriented attacks, through enforcing the execution semantics of control programs in the cyber-physical domain. Specifically, our program anomaly detection enforces the consistency among control decisions, values of data variables in control programs, and the physical environments. Our main technical contributions are summarized as follows.

- We describe a new security methodology, named *Orpheus*, that leverages the event-driven nature in characterizing control program behaviors. We present a general method for reasoning cyber-physical execution semantics of a control program, including the event identification and dependence analysis. We present a new event-aware finite-state automaton (*e*FSA) model to detect anomalous control program behaviors particularly caused by data-oriented attacks. By enforcing runtime cyber-physical execution semantics, *e*FSA detects subtle data-oriented exploits when physical event are inconsistent with the corresponding event-dependent state transitions. While our exposition of *Orpheus* is on an FSA model at the system call level, the design paradigm of *Orpheus* can be used to augment many existing program behavior models, such as the n-gram model [59] or HMM model [60].
- We implement a proof-of-concept prototype on Raspberry Pi platforms, which have emerged as popular devices for building CPS/IoT applications [11, 37, 52]. Our prototype features: i) A gray-box FSA model that examines the return addresses on the stack when system calls are made, and thus significantly increases the bar for constructing evasive mimicry attacks. ii) An LLVM-based event dependence analysis tool to extract event properties from programs and correlate the physical context with runtime program behaviors, which we refer to as *cyber-physical execution semantics*. iii) A near-real-time anomaly detector using named pipes, with both local and distributed event verifiers to assess the physical context.
- We conduct a thorough evaluation of *e*FSA's performance through three real-world embedded control applications. Results show that our approach can successfully detect different runtime data-oriented attacks reproduced in our experiments. Our prototype of the runtime anomaly detector takes ~0.0001s to check each state transition in *e*FSA model, ~0.063s for the local event verification, and ~0.211s for the distributed event verification.

The focus of this paper is on providing new security capabilities by enforcing cyber-physical execution semantics in defending against data-oriented attacks. Our design is a general approach for event-driven embedded control systems. In Sec. 7, we discuss in-depth practical deployment issues, including program anomaly detection as a service, implementation on bare-metal devices and programmable logic controllers (PLCs), and possible low overhead tracing with real-time requirements.

2 MODEL AND DESIGN OVERVIEW

In this section, we use examples to illustrate our new detection capabilities, and describe the threat model of this work. Then, we present the design overview of the *e*FSA model.

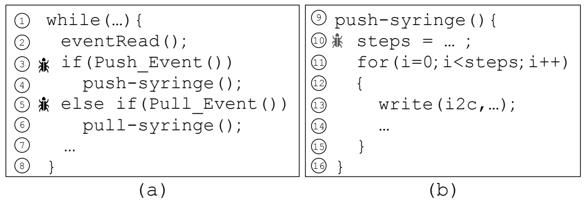
2.1 New Detection Capabilities

Our new detection capability is detecting data-oriented attacks in control programs, including hijacked for/while-loops or conditional branches. These stealthy attacks alter the underlying control program's behaviors without tampering control-flow graphs (CFGs). We illustrate our new detection capabilities using a smart syringe pump as an example [3]. The control program reads humidity sensor values as well as takes remote user commands, and translates the input values/commands into control signals to its actuator. Partial code is shown in Fig. 1. Our approach reasons about control programs' behaviors w.r.t physical environments, and is able to detect the following attacks:

- *Attacking control branch.* An attack affecting the code in Fig. 1(a) may trigger `push-syringe` or `pull-syringe` regardless of physical events or remote requests. It corrupts control variables that result in event function `Push_Event` or `Pull_Event` returning `True` (in lines 3 or 5). Such an attack leads to unintended but valid control flows.
- *Attacking control intensity.* An attack affecting the code in Fig. 1(b) may corrupt a local state variable (*e.g.*, `steps` in line 10) that controls the amount of liquid to dispense by the pump. An attack may cause the syringe to overpump than what is necessary for the physical environment. Range-based anomaly detection would not work, as the overwritten variable may still be within the permitted range (but incompatible with the current physical context). Such an attack (*i.e.*, manipulating the control loop iterations) does not violate the program's CFG either.

Existing solutions cannot detect these attacks, as the detection does not incorporate events and cannot reason about program behaviors w.r.t. physical environments. C-FLAT [11], which is based on the attestation of control flows and a finite number of permitted execution patterns, cannot fully detect these attacks. Similarly,

[3]https://hackaday.io/project/1838-open-syringe-pump



```
while(…){
  eventRead();
  if(Push_Event())
    push-syringe();
  else if(Pull_Event())
    pull-syringe();
  …
}
```

```
push-syringe(){
  steps = … ;
  for(i=0;i<steps;i++)
  {
    write(i2c,…);
    …
  }
}
```

Figure 1: Two examples of data-oriented software exploits in a real-world embedded control application. An attacker could purposely (a) trigger control actions by manipulating the return value of `Push_Event` or `Pull_Event`, and (b) manipulate the number of loop iterations in `push-syringe` without violating the control program's CFG.

recent frequency- and co-occurrence-based anomaly detection approaches (*e.g.*, global trace analysis [48] and system call frequency distribution (SCFD) [62]) cannot detect such either type of attacks, as their analyses do not model runtime cyber-physical context dependencies.

2.2 System Model and Definition of Events

Fig. 2 shows an abstract view of the event-driven CPS/IoT system architecture, which is also in line with the architecture of modern Industrial Control Systems (ICS)[4]. In general, it is composed of the following components: 1) a physical process (*e.g.*, industrial plant or smart home); 2) sensors that measure the physical environment; 3) actuators that trigger physical changes in response to control commands sent by the control program; 4) control programs running on embedded devices that supervise and control physical processes by taking sensory data as input and making local control decisions; 5) a remote control server (which is optional), letting users remotely monitor and control the physical process. A control program communicates with the physical process through sensors and actuators, where physical environments are sensed and events (*e.g.*, coming from the environment or emitted by other entities) are detected, and then actuation tasks are executed through a set of actuators.

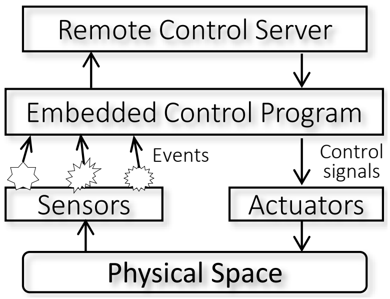


Figure 2: An abstract view of the embedded control system architecture in event-driven CPS/IoT applications.

Without loss of generality, we define two types of events in control programs: binary events and non-binary events.

- Binary events return either `True` or `False`, which are defined in terms of pre-specified status changes of physical environments and provide notifications to the control program (*e.g.*, `Push_Event` or `Pull_Event` in Fig. 1). Such events are commonly pre-defined and used in IoT's trigger-action programming ("if, then") model [32, 54].
- Non-binary events correspond to the sensor-driven control actions within a for/while loop, *e.g.*, sensor values affect the amount of control operations of `push-syringe` in Fig. 1. It is challenging to identify non-binary events since they are not explicitly declared in control programs.

Embedded devices (*a.k.a.* field devices) in CPS/IoT applications are situated in the field, where their operating systems are typically embedded Linux/Windows variants [46] or PLC firmware [28]. Traditionally, many of these embedded control systems were not considered prominent attack targets due to their isolation from potential attack sources. However, the historical isolation has begun to break down as more and more embedded devices are connected to business networks and the Internet in the trend of IoT, making control programs running on embedded devices (as well as the remote control server) increasingly vulnerable [46].

2.3 Attack Model and Assumptions

In this paper, we make the following security assumptions:

- *Capabilities of the adversary.* We assume that the adversary has successfully authenticated embedded devices (or the control server) under her control to the local network, and is able to launch runtime software exploits which may be unknown or known but unpatched at the time of intrusion. We are not concerned how attackers gained entry into the devices and launch different attacks, but focus on uncovering abnormal program execution behaviors after that [36]. This is a typical assumption in existing anomaly detection works.
- *CPS platform.* We assume the initial state (*i.e.*, the training stage) of the application is trustworthy, which is a general requirement of most behavior-based intrusion detection systems [62]. We also assume the runtime monitoring module is trusted and cannot be disabled or modified. This assumption is reasonable because it can be achieved by isolating the monitoring module from the untrusted target program with hardware security support such as Intel's TrustLite or ARM's TrustZone [11]. At the time of detection, the user space is partially or fully compromised, but the operating system space has not been fully penetrated yet, and thus it is still trusted [65].
- *Our focus.* We focus our investigation on runtime software exploits, and thus sensor data spoofing attacks in the physical domain [53] are out of the scope. We assume sensor measurements are trustable. We limit our attention to data-oriented attacks that involve *changes of system call usage*. Other data-related attacks that do not impact observable program behavior patterns (*e.g.*, modification of non-decision making variables) are beyond the scope of this work. System call can be used as an ideal signal for detecting potential intrusions, since a compromised program can generally cause damage to the victim system only by exploiting system calls [21]. Despite system call based monitoring is widely used for detecting compromised programs, we aim

[4]In industrial control domain, the control program is often referred to as control logic, and the firmware on PLC (*i.e.*, field device) acts as a kind of operating system [28].

at developing a CPS/IoT-specific anomaly detection system by augmenting an existing program behavior model with physical context awareness.

2.4 Design Overview and Workflow

An anomaly detection system is normally composed of two stages: training (where program behavior models are built based on normal program traces) and testing (where a new trace is compared against the model built in the training phase). Fig. 3 shows the steps for constructing the *e*FSA program behavior model in our design. In particular, to capture the cyber-physical context dependency of control programs, our training stage encompasses both static program analysis and dynamic profiling.

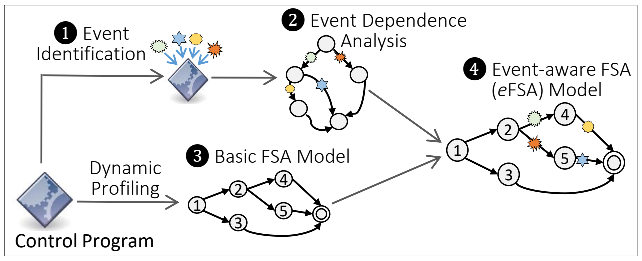


Figure 3: *e*FSA model construction in the training phase. This workflow can be generalized to non-FSA anomaly detection frameworks (*i.e.*, augmenting an existing program behavior model with contextual integrity).

There are four main steps in the training phase. We first identify both binary events and non-binary events involved in the control program (❶). After that, we perform the event dependence analysis to generate an event-annotated CFG (❷), which identifies event triggered instructions/statements of the program (corresponding to binary events), and control intensity loops (corresponding to non-binary events). Then, we construct the basic finite-state automaton (FSA) model based on dynamic profiling (❸). Given an event-annotated CFG, we are able to identify the event-driven system call sequences. To detect control intensity anomalies, we conduct a control intensity analysis and associate the results with corresponding non-binary events. By augmenting the event-driven information over the basic FSA, we generate our event-aware FSA (*i.e.*, *e*FSA) for control program behavior modeling (❹).

The basic FSA model aims at detecting control-oriented attacks. Our main contribution lies in the event awareness enhancement based on the FSA model, which checks the consistency between runtime behavior and program execution semantics. In the testing phase, an anomaly is marked if there exists a state transition deviated from the automaton, or a mismatch between the physical context and program control-flow path.

3 EFSA CONSTRUCTION WITH STATIC ANALYSIS AND DYNAMIC PROFILING

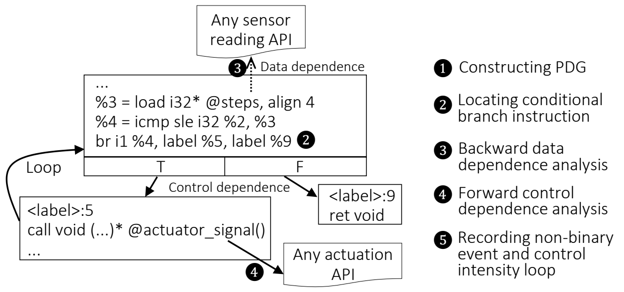
In what follows, we first present how we reason cyber-physical execution semantics of a control program through static analysis. Then, we describe details about how to build the *e*FSA model.

3.1 Event Identification

In order to discover the triggering relationship between external events and internal program control flows, we first identify what events are involved in a control program. For pre-defined binary events, it is not difficult to identify these events (*e.g.*, given event functions declared in an event library or header file, we scan the source code or executable binary). The main challenge is to identify i) non-binary events and ii) non-pre-defined binary events. Our LLVM-based [5] event identification algorithm can automatically extract these events and only requires knowledge of sensor-reading APIs and actuation APIs on the embedded system. They are pre-specified sources and sinks[5] in our static analysis.

According to the definition of a non-binary event in Sec. 2.2, it contains a loop statement (*e.g.*, for/while loop) in which sensor values affect the amount of control operations. Our key idea is to search for a loop statement that is data-dependent on any sensor-reading API, and at least an actuation API is control-dependent on this loop statement. The search is performed through backward data dependence analysis and forward control dependence analysis.

A more specific step-by-step description of our event identification is illustrated in Fig. 4 using a C-based control program as an example. ❶ Constructing PDG: We first obtain the LLVM Intermediate Representation (IR) of a control program using the `Clang` compiler [5], and construct the program dependence graph (PDG) at basic block level[6], including both data and control dependencies. ❷ Locating conditional branch instruction with a loop: We search the conditional "br" instruction, which takes a single "i1" value and two "label" values. ❸ Backward data dependence analysis: We conduct backward inter-procedural dataflow analysis to find any prior data dependence on sensor-reading APIs. ❹ Forward control dependence analysis: We then conduct forward inter-procedural control-dependence analysis on the true branch of the conditional instruction to find actuation APIs (*e.g.*, APIs in WiringPi library or functions writing GPIO pins [4]). ❺ Recording the identified non-binary event and control intensity loop: This is the output of the event identification module.



Any sensor reading API
❸ Data dependence
...
%3 = load i32* @steps, align 4
%4 = icmp sle i32 %2, %3
br i1 %4, label %5, label %9 ❷
Loop
T
F
Control dependence
<label>:5
call void (...)* @actuator_signal()
...
<label>:9
ret void
Any actuation API
❹
❶ Constructing PDG
❷ Locating conditional branch instruction
❸ Backward data dependence analysis
❹ Forward control dependence analysis
❺ Recording non-binary event and control intensity loop

Figure 4: An example of identifying non-binary events

We also design a similar procedure for identifying non-predefined binary events. An example of such event is when the temperature exceeds a user-designated value, an event predicate returns

[5]Source and sink are terms in a dataflow analysis. The source is where data comes from, and the sink is where it ends in a program [45].

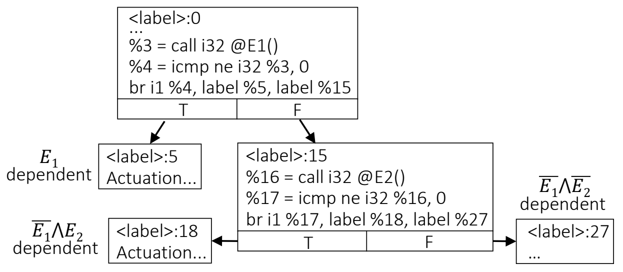
[6]In program analysis, a basic block is a linear sequence of instructions containing no branches except at the very end.

True. In this procedure, we search for the conditional branch either "br" or "switch" instruction without a loop, and then perform the same data/control dependence analysis. In particular, we need to analyze both true and false branches of a "br" instruction, because both branches may contain control actions and we also consider the not-happening case (*i.e.*, the branch without triggering any control action) as an implicit event.

3.2 Event Dependence Analysis

Our event dependence analysis generates an event-annotated CFG, *i.e.*, approximating the set of statements/instructions that connect events and their triggered actions. During the event identification, we identify individual events that are involved in a control program. We directly associate a non-binary event with its control intensity loop. A challenge arises when dealing with nested binary events.

We address the nested events challenge using a bottom-up approach for recursive searching for event dependencies. Given a binary-event triggered basic block, we backward traverse all its control dependent blocks until reaching the root, and extract corresponding branch labels (*i.e.*, True or False).


```
<label>:0
...
%3 = call i32 @E1()
%4 = icmp ne i32 %3, 0
br i1 %4, label %5, label %15
```
```
<label>:5
Actuation...
```
```
<label>:15
%16 = call i32 @E2()
%17 = icmp ne i32 %16, 0
br i1 %17, label %18, label %27
```
```
<label>:18
Actuation...
```
```
<label>:27
...
```

T
F
E_1 dependent
$\overline{E_1} \wedge E_2$ dependent
$\overline{E_1} \wedge \overline{E_2}$ dependent

Figure 5: Event dependence analysis for nested events

Fig. 5 illustrates an example of our event dependence analysis for nested binary events. Block 18 is control dependent on Block 15 in the True branch of E_2 (called true-control-dependent). By backward traversing the control dependence graph, we find Block 15 is further false-control-dependent on E_1 in Block 0. Then, we know Block 18 is control dependent on a composite event $[\overline{E_1} \wedge E_2]$. In this example, we also find event dependencies for Blocks 5 and 27. Finally, we transform instruction-level event dependencies in LLVM IR to statement-level dependencies in source code with line numbers, which are the outputs of the event dependence analysis.

In addition to the static analysis approach, an alternative for event dependence analysis is using dynamic slicing [63], which identifies statements triggered by a particular event during multiple rounds of program executions. It is worth mentioning that our event identification and dependence analysis is a general approach for reasoning cyber-physical execution semantics, independent of specific program anomaly detection models.

3.3 Formal Description of *e*FSA

In this and the next few sections, we describe a specific FSA-based anomaly detection model, which captures the event-driven feature of control programs to detect evasive attacks.

We construct the finite-state automaton (FSA) [47] model, which is based on tracing the system calls and program counters (PC) made by a control program under normal execution. Each distinct PC (*i.e.*, the return address of a system call) value indicates a different state of the FSA, so that invocation of same system calls from different places can be differentiated. Each system call corresponds to a state transition. Since the constructed FSA uses memory address information (*i.e.*, PC values) in modeling program behaviors (called the gray-box model), it is more resistant to mimicry attacks than other program models [26, 50]. In an execution trace, given the k_{th} system call S_k and the PC value pc_k from which S_k was made, the invocation of S_k results in a transition from the previous state pc_{k-1} to pc_k which is labelled with S_{k-1}. Fig. 6(a) shows a pictorial example program, where system calls are denoted by $S_0, \ldots, S_6$, and states are represented by integers (*i.e.*, line numbers). Suppose we obtain three execution sequences, $\frac{S_0}{1}\frac{S_1}{3}\frac{S_2}{6}\frac{S_3}{7}\frac{S_2}{6}\frac{S_3}{7}\frac{S_5}{10}\frac{S_6}{11}$, $\frac{S_0}{1}\frac{S_1}{3}\frac{S_4}{9}\frac{S_4}{9}\frac{S_5}{10}\frac{S_6}{11}$, and $\frac{S_0}{1}\frac{S_1}{3}\frac{S_5}{10}\frac{S_6}{11}\frac{S_1}{3}\frac{S_5}{10}\frac{S_6}{11}$, the learnt FSA model is shown in Fig. 6(b), where each node represents a state and each arc represents a state transition.

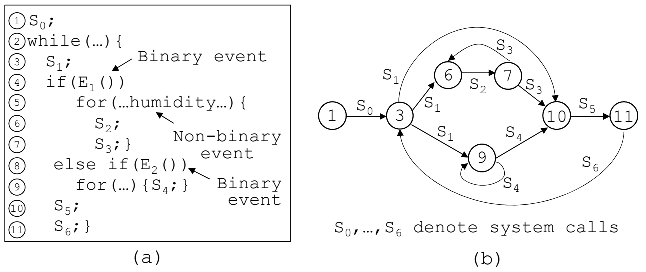
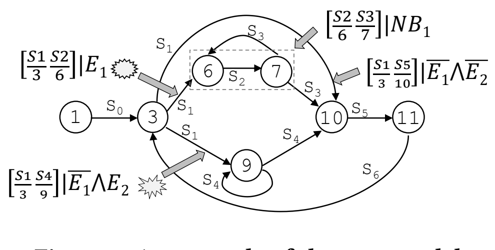

```
① S0;
② while(…){
③   S1;
④   if(E1())
⑤     for(…humidity…){
⑥       S2;
⑦       S3;}
⑧   else if(E2())
⑨     for(…){S4;}
⑩   S5;
⑪   S6;}
```

Figure 6: System call based finite-state automaton (FSA) model: (a) an example program; (b) the corresponding FSA model.

We formally define the *e*FSA model as a six-tuple: $(S, \Sigma, s_0, F, E, \delta)$. S is a finite set of states which are PC values, and Σ is a finite set of system calls (*i.e.*, input alphabet). s_0 is an initial state, and F is the set of final states. E represents a finite set of external events, which can affect the underlying execution of a control program. δ denotes the transition function mapping $S \times \Sigma \times E$ to S. Note that a state transition may come with multiple physical events (referred to as a composite event). Thus, the input alphabet can be expressed as a cartesian product: $E = E_1 \times E_2 \times \cdots \times E_n$, where the input E consists of n concurrent physical events. In particular, we consider the non-occurrence (not-happening) of one or more events as an implicit event in *e*FSA.



$[\frac{S2}{6}\frac{S3}{7}]|NB_1$
$[\frac{S1}{3}\frac{S2}{6}]|E_1$
$[\frac{S1}{3}\frac{S5}{10}]|\overline{E_1} \wedge \overline{E_2}$
$[\frac{S1}{3}\frac{S4}{9}]|\overline{E_1} \wedge E_2$
1
3
6
7
9
10
11
S_0
S_1
S_1
S_1
S_2
S_3
S_3
S_4
S_4
S_5
S_6

Figure 7: An example of the *e*FSA model

Fig. 7 shows an example of *e*FSA model corresponding to the FSA example in Fig. 6, where an event dependent transition is labeled by "$[\frac{System\ Call}{PC}]$|Events". In this example, there are two binary events and one non-binary event. We identify binary-event dependent state transitions $[\frac{S_1}{3}\ \frac{S_2}{6}]|E_1$, $[\frac{S_1}{3}\ \frac{S_4}{9}]|E_2$, and a non-binary-event dependent control intensity loop $[\frac{S_2}{6}\ \frac{S_3}{7}]|NB_1$. It also contains an implicit event dependent transition $[\frac{S_1}{3}\ \frac{S_5}{10}]|(\overline{E_1} \wedge \overline{E_2})$. *e*FSA expresses causal dependencies between physical events and program control flows. By checking execution semantics at runtime, *e*FSA improves the robustness against data-oriented attacks by increasing the difficulties that an attack could bypass the anomaly detection.

3.4 Security Policies in *e*FSA

Our *e*FSA model extends FSA with external context constraints, where event-dependent state transitions in FSA are labeled with event constraints. To construct an *e*FSA, we need to identify event dependent state transitions in FSA. Towards this end, we apply the event dependence analysis results (described in Sec. 3.1 and 3.2) to transform statement-level dependencies in source code to the state transition dependencies in FSA. Such a mapping might be achieved through static analysis, *e.g.*, passing over the parse tree to search for system call invocations. However, a static analysis based approach requires the modifications of gcc compiler or system call stubs, and even requires hand-crafted modifications for library functions [34, 57]. In *e*FSA, we adopt a dynamic profiling based approach to discover event dependent state transitions, where details are introduced in Sec. 5.

For state transitions that are dependent on binary events, the cyber-physical policy enforcement is to make sure these binary events return the ground truth values. For control intensity loops that are dependent on non-binary events, we enforce security policies through a control intensity analysis, which models the relationship between the observable information in cyber space (*i.e.*, system calls) and sensor values in physical space. *e*FSA then enforces the policy that the observed control intensity should be consistent with the corresponding sensor measurements.

3.5 Control Intensity Analysis

The main challenge for detecting runtime control intensity anomalies lies in that, given system call traces of a control program, we need to map the control intensity to its reflected sensor measurements, where only the number of loop iterations in a control intensity loop is available. To this end, we first obtain the number of system calls invoked in each loop iteration. Then, we model the relationship between sensor measurements and the amount of system calls in a control intensity loop through regression analysis.

Execution Window Partitioning and Loop Detection: Typically, control programs monitor and control physical processes in a continuous manner, where the top-level component of a program is composed of an infinite loop. For instance, an Arduino program [1] normally consists of two functions called `setup()` and `loop()`, allowing a program consecutively controls the Arduino board after setting up initial values. We define an *execution window* as one top-level loop iteration in a continuous program, and a *behavior instance* as the program activity within an execution window. The term execution window is equivalent to the *scan cycle* in industrial control domain [37]. We partition infinite execution traces into a set of behavior instances based on the execution window. The underlying FSA model helps identify loops since it inherently captures program loop structures. We first identify the starting state in the top-level loop of a FSA. Then, once a top-level loop back edge is detected, a behavior instance is obtained.

Regression Analysis: The purpose of the regression analysis is to quantify the relationship between sensor measurements and system call amount in a control intensity loop. Given the number of system calls invoked in each loop iteration, one straightforward approach is through manual code analysis. In this work, we present an approach for automating this process. During the identification of non-binary events in Sec. 3.1, we know what sensor types (*i.e.*, sensor reading APIs) are involved in a control intensity loop. In the training phase, we collect normal program traces together with the corresponding sensor values. Then, we perform a simple regression analysis to estimate the relationship between the system call amount (*i.e.*, outcome) and sensor measurements (*i.e.*, explanatory variables) for each control intensity loop. For example, suppose a control intensity loop is triggered by the change of humidity sensor value (details are in Sec. 6.4). We observe that an increase of humidity results in more iterations of the control intensity loop, where each loop iteration incurs 3 system calls. Thus, we can reversely derive the changes of physical environment by observing the number of iterations in a control intensity loop.

4 EFSA-BASED DETECTION

In this section, we present how an *e*FSA-based anomaly detector detects anomalies particularly caused by data-oriented attacks, and discuss about the design choices of event verification.

4.1 Runtime Monitoring and Detection

Our anomaly detector traces system calls as well as the corresponding PC values during the execution of a control program. The anomaly detection is composed of two steps: i) state transition integrity checking against the basic FSA model, and ii) event consistency checking against the event verification in the *e*FSA-based anomaly detector, which is our new contribution.

- **Event-independent state transition.** For each intercepted system call, we check if there exists an outgoing edge labelled with the system call name from the current state in FSA. If not, an anomaly is detected. If the current state transition is not event-dependent, we move the current state of the automaton to the new state. This basic state-transition checking has been shown to be effective against common types of control-oriented attacks (*e.g.*, code injection attacks or code-reuse attacks [24]) which violate control flow integrity of the model.
- **Event-dependent state transition.** In case of an event dependent state transition according to *e*FSA, we first perform the above basic state-transition checking. More importantly, with the help of the event verification (discussed in Sec. 4.2), we then check the consistency between the runtime execution semantics and program's behavior, *i.e.*, whether a specific physical event associated with this event-dependent state transition is observed in the physical domain. This step can detect stealthy data-oriented attacks that follow valid state transitions but are incompatible

with the physical context. Another important aspect is the selection of event checkpoints. To avoid redundant checking, we set the checkpoint for a binary event at its first event-dependent state transition. For a non-binary event, we perform the event checking after it jumps out of the control intensity loop.

4.2 Event Verification Strategies

The objective of event verification is to detect event spoofing caused by runtime data-oriented software exploits. Event verification is highly application specific, and it is actually orthogonal to the *e*FSA model itself. We describe several possible approaches for verifying physical context.

- *Local event verification:* which is able to detect the inconsistency between program runtime behavior and cyber-physical execution semantics. For example, the monitor re-executes a binary-event function to confirm the occurrence of the event. To detect control intensity anomalies, the monitor retrieves sensor measurements and compares them against the derived sensor values from system call traces. There may exist false positives/negatives due to sensor's functional failures in practice.
- *Distributed event verification:* which assesses the physical context by exploiting functionally and spatially redundancy of sensors among co-located embedded devices. Since sensor data normally exhibit spatio-temporal correlation in physical environments, it increases the detection accuracy by involving more event verification sources.
- *Physical model based verification:* complementary to the runtime event verification, cyber-physical inconsistency may be detected based on physical models [55]. For example, one may utilize fluid dynamics and electromagnetics as the basic laws to create prediction models for water system [30] and power grid [35]. Based on the prediction models and predefined threat constraints, these methods can then check whether the predicted environment values are consistent with a control system's behavior.

5 IMPLEMENTATION

To demonstrate the feasibility of our approach, we have implemented a prototype with around 5K lines in C/C++, Bash and Python codes, including the trace collection and preprocessing, event identification and dependence analysis, *e*FSA model construction, and runtime anomaly detection modules. Our prototype uses multiple off-the-shelf tools and libraries in Linux.

We choose Raspberry Pi 2 with Sense HAT as the main experimental platform, which is a commonly used platform for building embedded control applications [11, 37, 52]. Sense Hat, an add-on board for Raspberry Pi, provides a set of environmental sensors to detect physical events including pressure, temperature, humidity, acceleration, gyroscope, and magnetic filed. During the training phase, we collect program traces on Raspberry Pi and perform the *e*FSA model construction on a Linux Desktop (Ubuntu 16.04, Intel Xeon processor 3.50GHz and 16GB of RAM). In the testing phase, the anomaly detector is deployed on Raspberry Pi to detect runtime control-based or data-oriented attacks. As a special case, we conduct experiments for post-mortem analysis of anomalous behaviors on a commercial drone to demonstrate how *e*FSA can be applied to network event-triggering scenarios (where details can be found in Sec. 6). In the following, we present key implementation aspects in our prototype.

Dynamic Tracing. We use the system tool `strace-4.13` to intercept system call of a running control program. To obtain the PC value from which a system call was invoked in a program, we need to go back through the call stacks until finding a valid PC along with the corresponding system call. We compile `strace` with `-libunwind` support, which enables stack unwinding and allows us to print call stacks on every system call.

```
[76eb989c] write(1, "Start\n", 6) = 15
>/lib/arm-linux-gnueabihf/libc-2.19.so(__write+0x1c) [0xc289c]
>/lib/arm-linux-gnueabihf/libc-2.19.so(_IO_file_write+0x48) [0x6b008]
>/lib/arm-linux-gnueabihf/libc-2.19.so(_IO_file_setbuf+0xd4) [0x6a4a8]
>/lib/arm-linux-gnueabihf/libc-2.19.so(_IO_do_write+0x18) [0x6c038]
>/lib/arm-linux-gnueabihf/libc-2.19.so(_IO_file_overflow+0xf4) [0x6c408]
>/lib/arm-linux-gnueabihf/libc-2.19.so(__overflow+0x20) [0x6cf14]
>/lib/arm-linux-gnueabihf/libc-2.19.so(_IO_puts+0x140) [0x615b8]
>/home/pi/Solard(main+0x20) [0x43c]
```

Figure 8: An example of using `strace` tool with stack unwinding support, where call stacks are printed out with the system call.

It is worth mentioning that our model works in the presence of Address Space Layout Randomization (ASLR), which mitigates software exploits by randomizing memory addresses, as the low 12 bits of addresses are not impacted by ASLR (PC values can be easily aligned among different execution traces of a program). Fig. 8 shows an example of using `strace` tool with stack unwinding support. In this example, we use the PC value of relative address `0x43c` for the `write` system call. As a result, system calls that are triggered from different places in a program will be associated with different PC values, which enables the FSA model to accurately capture a program's structures (*e.g.*, loops and branches).

Event Identification and Dependence Analysis. Our event identification and dependence analysis tool is implemented within the Low Level Virtual Machine (LLVM)[7] compiler infrastructure, based on an open source static slicer[8] which builds dependence graph for LLVM bytecode. An advantage of using LLVM-based event dependence analysis is that, our tool is compatible with multiple programming languages since LLVM supports a wide range of languages. Our event identification module identifies the line numbers in source code where an event is involved. Then, the event dependence analysis outputs the line numbers of event dependent statements. After discovering statement-level (*i.e.*, instruction-level) event dependence, we next identify event-dependent state transitions (*i.e.*, system call level) in FSA. By using the `addr2line` tool, we could map line numbers and file names to return addresses (*i.e.*, PC values) that are collected in dynamic profiling phase. Subsequently, we augment the event-driven information over the underlying FSA, and finally construct the *e*FSA model.

Anomaly Detector with Event Verification. In our prototype, we implement a proof-of-concept near-real-time anomaly detector using named pipes on Raspberry Pi, including both local and distributed verifications (corroboration with single or multiple external sources). We develop a sensor event library for Raspberry Pi Sense Hat in C code, based on the sensor reading modules in `experix`[9] and `c-sense-hat`[10]. The event library reads pressure

[7] http://llvm.org/
[8] https://github.com/mchalupa/dg
[9] http://experix.sourceforge.net/
[10] https://github.com/davebm1/c-sense-hat

and temperature from the LPS25H sensor, and reads relative humidity and temperature from the HTS221 sensor, with maximum sampling rates at 25 per second. Our local event verifier calls the same event functions as in the monitored program, and locally check the consistency of event occurrence. In the distributed event verifier, we deploy three Raspberry Pi devices in an indoor laboratory environment. We develop a remote sensor reading module which enables one device to request realtime sensor data from neighbouring devices via the sockets communication.

6 EXPERIMENTAL VALIDATION

We conduct three real-world case studies of embedded control and monitoring systems, and evaluate *e*FSA's detection capability against runtime data-oriented attacks. Our experiments aim to answer the following questions:

- What is the runtime performance overhead of *e*FSA (Sec. 6.2)?
- Whether *e*FSA is able to detect different data-oriented attacks (Sec. 6.3 and 6.4)? We also provide a video demo to demonstrate *e*FSA's detection capability[11].

6.1 Case Studies of Embedded Control Systems

Solard[12]**.** It is an open source controller for boiler and house heating system that runs on embedded devices. The controller collects data from temperature sensors, and acts on it by controlling relays via GPIO (general purpose input/output) pins on Raspberry Pi. Control decisions are made when to turn on or off of heaters by periodically detecting sensor events. For example, `CriticalTempsFound()` is a pre-defined binary event in Solard. When the temperature is higher than a specified threshold, the event function returns `True`.
SyringePump[13]**.** It was developed as an embedded application for Arduino platform. Abera *et al.* [11] ported it to Raspberry Pi. The control program originally takes remote user commands via serial connection, and translates the input values into control signals to the actuator. SyringePump is vulnerable since it accepts and buffers external inputs that might result in buffer overflows [11]. We modify the syringe pump application, where external inputs are sent from the control center for remote control, and environmental events drive the pump's movement. Specifically, in the event that the relative humidity value is higher than a specified threshold, the syringe pump movement is triggered. In addition, the amount of liquid to be dispensed is linearly proportional to the humidity value subtracted by the threshold. Such sensor-driven syringe pumps are used in many chemical and biological experiments such as liquid absorption measurement experiment.
AR.Drone 2.0 UAV[14]**.** The Parrot AR.Drone 2.0 is a remote controlled quadrocopter, where the control unit receives commands from the remote ground station, monitors and controls the system status to coordinate the flight. Rodday *et al.* [43] exploit security vulnerabilities of the AR.Drone to inject malicious packets and control the Drone. In our experiment, we reproduce the command spoofing attacks to AR.Drone.

[11] https://youtu.be/-VEjidSgGIc
[12] https://github.com/mrpetrov/solarmanpi
[13] https://github.com/control-flow-attestation/c-flat
[14] https://www.parrot.com/us/drones/parrot-ardrone-20-elite-edition

6.2 Training and Runtime Performance

In the training phase, we collect execution traces of Solard and SyringePump using training scripts that attempt to simulate possible sensor inputs of the control programs. By checking Solard and SyringePump's source codes, our training scripts cover all execution paths. Since AR.Drone allows a connection to the Telnet port which leads to a root shell, we are able to deploy the `strace` tool to collect system call traces of the UAV control program. We collect execution traces of AR.Drone by running it using the public testing script[15], which sequentially sends different control commands to the drone. The control program (*i.e.*, `program.elf`) forks 31 child processes, where we separate system call traces for each process.

We first measure the time taken for training models in our prototype, where the main overhead comes from the event dependence analysis. In AR.Drone, the system call types involved in the process that handles remote commands are quite limited and the program logic is rather simple. Thus, we can easily construct the corresponding *e*FSA model by taking advantage of network protocol interactions (*i.e.*, network API semantics [64]).

	Event Dependence Analysis	
	Desktop Computer	Raspberry Pi 2
Solard	0.745s	109.975s
SyringePump	0.0035s	1.726s

Table 1: Average delay overhead in training phase

Table 1 illustrates *e*FSA's program analysis overhead in the training phase. For comparison purpose, we deploy the LLVM toolchain and our event dependence analysis tool on both Raspberry Pi and Desktop Computer (Intel Xeon processor 3.50GHz and 16GB of RAM). From Table 1, Raspberry Pi takes much longer time (more than 150 times) than desktop computer to complete the program dependence analysis task. It only takes 0.745s and 0.0035s for event dependence analysis of Solard (46.3 kb binary size) and SyringePump (17.7 kb binary size) on a desktop computer, respectively. Since Solard and SyringePump run in a continuous manner and thus generate infinite raw traces. The model training overhead is measured by how much time it takes for training per MByte raw trace. Results show that it takes less than 0.2s to process 1 MByte traces on the desktop computer. The number of states in Solard's and SyringePump's *e*FSA is 34 and 65, respectively.

Delay (Raspberry Pi 2)	Mean	Standard Deviation
FSA State Transition Checking	0.00013293s	0.00004684s
Local Event Verification	0.06279120s	0.00236999s
Distributed Event Verification	0.21152867s	0.03828739s

Table 2: Runtime overhead in the monitoring phase

Next, we measure the performance overhead incurred by *e*FSA's anomaly detector on Raspberry Pi. The system call tracing overhead has no difference between FSA and *e*FSA, incurring 1.5x∼2x overhead in our experiments. Table 2 reports the runtime detection latency results. The average delay for each state transition (*i.e.*, each intercepted system call) checking out of more than 1000 runs is around 0.0001s. It takes 0.063s on average to perform the local event checking. The end-to-end latency for the distributed event checking from each co-located device can be broken down into

[15] https://github.com/felixge/node-ar-drone

two main parts: i) network communication around 0.042s, and ii) sensor reading delay around 0.0582s. In our experiment, we deploy two co-located devices, and thus the total distributed event checking delay is around 0.212s. It is expected that the overhead of distributed event checking is linearly proportional to the number of event verification sources.

6.3 Detecting Attacks on Control Branch

In this experiment, we evaluate *e*FSA's security guarantees against control branch attacks. In Solard, our buffer overflow attack manipulates the temperature sensor values to maliciously prevent the heater from being turned off. This cyber-physical attack is similar to the recent real-world German steel mill attack [3], which may result in a blast furnace explosion. In this experiment, we attach the Raspberry Pi on an electric kettle (*i.e.*, 1-Liter water boiler). The control program keeps monitoring temperature values. When the temperature is lower than $50°C$, it turns on the heater. And when the temperature is higher than $60°C$, where `CriticalTempsFound()` is supposed to return `True`, it turns off the heater. In the monitoring phase, when we detect an event-dependent state transition in *e*FSA model, the local event verifier performs event consistency checking. (Details of SyringePump case study against control branch attacks can be found in our video demo)

Fig. 9 illustrates an instance of the Solard experiment. We corrupt the temperature sensor values in the range of $40{\sim}45°C$, which falsifies the return value of `CriticalTempsFound()` to be always `False`. In every scan cycle, *e*FSA observes a state transition dependent on the not-happening of `CriticalTempsFound()` (*i.e.*, an implicit event), and thus the event verifier checks the instantaneous temperature value. In our experiment, because the Raspberry Pi does not physically interact with the electric kettle, the ground truth temperature keeps increasing up to more than $80°C$ in Fig. 9. However, *e*FSA successfully raises an alarm at the first moment when it finds a mismatch between the execution semantics (temperature exceeding $60°C$) and program behavior.

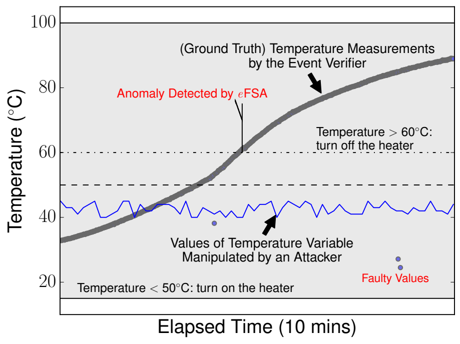
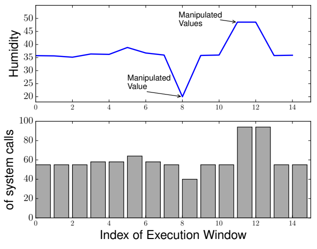


Figure 9: An instance of Solard experiment

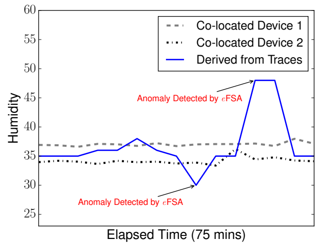
We did encounter sensor measurement failures, *e.g.*, isolated dots as shown in Fig. 9. On average, the false sensor measurement rate is lower than 1% in our experiments. This means that the detection rate and false positive/negative rate would depend on sensors' functional reliability in practice. Existing methods, such as data fusion [15] can be applied to enhance the detection accuracy.

6.4 Detecting Attacks on Control Intensity

In this experiment, we demonstrate that *e*FSA is able to detect control intensity attacks with only system call traces. In SyringePump, we set the threshold that triggers the movement of syringe pump to be $30rH$. Our attack corrupts the humidity sensor value based on a buffer overflow vulnerability [11] and thus could drive the movement of syringe pump without receiving an external event or environmental trigger (*i.e.*, regardless the relative humidity value is higher than $30rH$). Meanwhile, the corrupted humidity value also determines the amount of liquid to be dispensed, which equals to the humidity value subtracted by $30rH$. In the training phase, through control intensity analysis, we know the number of system calls with no event occurrence is 40 per scan cycle, and each loop iteration (*i.e.*, dispensing a unit of liquid) in the control intensity loop corresponds to 3 system calls.



(a) Humidity and system call traces



(b) *e*FSA's detection

Figure 10: An instance of SyringePump experiment with a sampling rate of 5 minutes

Fig. 10(a) shows the value changes of the humidity variable and system call amount per scan cycle of SyringePump. The normal humidity value fluctuates between 34 rH and $38rH$. As a result, the amount of liquid to be dispensed is subsequently changed, which is reflected by the number of system calls in each control loop. We manipulate the humidity values to be $20rH$ and $48rH$, respectively. In the monitoring phase, by observing the number of system calls in each control loop, we can reversely derive the changes of physical environment based on our control intensity regression model as shown in Fig. 10(b). In this test, if the difference between the derived value and the sampled average value from event verifier is larger than $3rH$, we consider it an anomaly. By checking the humidity measurements from two co-located devices (*i.e.*, denoted as devices 1 and 2), our distributed event verifier detects that the program's runtime behaviors are incompatible with physical contexts. Thus, *e*FSA successfully detects the control intensity attacks.

From Sec 6.3 and Sec. 6.4, we demonstrate that enforcing cyber-physical execution semantics in control-program anomaly detection is effective to detect both types of data-oriented attacks. As long as the current execution context is incompatible with the observed program state transitions, *e*FSA is able to detect potential anomalies.

6.5 Generalization of Events and Detecting Command Injection Attacks

Control programs running on embedded devices may receive network events from the control center, and then execute actuation tasks. Though *e*FSA mainly detects software-exploit based environmental event spoofing, we demonstrate it is also applicable to network event-triggering scenarios by conducting an experiment on AR.Drone. We consider each type of control command as a specific event, and the *e*FSA model is augmented with command events. To detect false command injection attacks, *e*FSA checks the consistency of system call traces at a UAV and its ground control station (GCS), ensuring their system call invocations conforming to the network API semantics [64]. We use a laptop (as an attacker) to send fake control commands to the AR.Drone. Meanwhile, we collect system calls on both Drone and GCS. The event verifier could find inconsistencies between *e*FSA's state transitions and network events (*e.g.*, `sendto` or `recvfrom`) retrieved from GCS, and thus detect this type of attacks. In this experiment, we do not intend to use *e*FSA to raise an alarm at the time of intrusion, instead we aim at detecting the anomalous behavior as a post-mortem analysis.

7 DEPLOYMENT DISCUSSION

Although our work is focused on providing new security capabilities in control-program anomaly detection against data-oriented attacks, in this section, we examine the limitations of our implementation and discuss how our method can be deployed in the near future.

Anomaly Detection as a Service: Embedded devices are resource-constrained compared with general-purpose computers. To reduce detection overhead, the anomaly detection may be performed at a remote server. We envision deployment involving partnerships between hardware vendors and security service providers (similar to ZingBox IoT Guardian [8]), where the security provider is given access to embedded platforms and helps clients to diagnose/confirm violations. The client-server architecture resonates with the remote attestation in embedded systems, which detects whether a controller is behaving as expected [11, 56]. For detection overhead reduction, the remote server may choose when and how frequently to send assessment requests to a control program for anomaly detection. It is also possible to selectively verify a subset of events, *e.g.*, only safety-critical events specified by developers are involved. While the event verifier implementation is not completely automated, our event identification and dependence analysis tool does automate a large portion of event code extraction and eases the developer's burden. We leave automatically generating event verification functions for the anomaly detector as an important part of our future work.

Embedded Devices Without OS: Our anomaly detection system works on the granularity of system calls and it leverages dynamic tracing facilities such as the `strace` tool, which requires the operating system support. An important reason behind our choice is that, the new generation of embedded control devices on the market are increasingly coming with operating systems [46, 52]. For example, Raspberry Pi devices with embedded Linux OS have been used as field devices in many CPS/IoT applications [6]. Linux-based PLCs for industrial control have emerged to replace traditional PLCs [7] for deterministic logic execution. However, embedded devices may still operate in bare-metal mode [11], where we can not utilize existing tracing facilities to collect system call traces. For traditional PLCs, our security checking can be added to the program logic. We can also apply the event checking idea to an anomaly detection system at the level of instructions. We may instrument the original control program with event checking hooks by rewriting its binary, *e.g.*, inserting hooks at the entry of event-triggered basic blocks. We consider it as the future work to extend our design paradigm for fine-grained anomaly detection with binary instrumentation.

Tracing Overhead and Time Constraints: Though system call traces are a common type of audit data in anomaly detection systems, we would like to point out that the conventional software-level system call tracing incurs unnegligible performance overhead to the monitored process [23]. It holds for time-insensitive embedded control applications, *e.g.*, smart home automation, but would be a technical challenge for time-sensitive applications. While we employ the user-space `strace` software to collect system calls in our prototype, tracing tools are orthogonal to our detection design. For performance consideration, alternative tracing techniques may be adopted in replacing `strace` to improve the tracing performance [48]. For example, it is possible to improve the performance for system call interposition by modifying the kernel at the cost of increased deployment effort. With the recently unveiled Intel's Processor Trace (PT) and ARM's CoreSight techniques, hardware tracing infrastructures are increasingly embedded in modern processors, which can achieve less than 5% performance overhead [13]. The recent work, Ninja [41], offers a fast hardware-assisted tracing on ARM platforms. The overhead of instruction tracing and system call tracing are negligibly small. Therefore, we anticipate that future tracing overhead will be significantly reduced as the hardware-assisted tracing techniques are increasingly used.

8 RELATED WORK

Our contribution in this work lies at the intersection of two areas: anomaly detection for embedded systems software and program behavior modeling. In this section, we briefly summarize related works in these two research areas.

8.1 Anomaly Detection for Embedded Software

Our work focuses on anomaly detection for embedded control programs with cyber-physical interactions, *e.g.*, in CPS/IoT applications. The majority of research efforts in this area thus far have concentrated on behavior model-based anomaly detection [55]. Since such embedded control system is an integration of cyber and physical components, previous anomaly detection works can be divided into two lines of research: based on i) behavior models of physical processes; or ii) behavior models of cyber programs/systems.

Due to the diversity of CPS/IoT applications, existing behavior models are proposed to detect anomalies for specific applications, such as smart grid [51], unmanned aerial vehicles [39], medical devices [14, 38], automotive [20, 44], industrial control process [16, 37, 55]. The idea of using physical models to define normal operations for anomaly detection is that, system states must follow immutable laws of physics. For example, authors in [58] derived a graph model to defeat false data injection attacks in the Supervisory Control and Data Acquisition (SCADA) system, by capturing internal relations among system variables and physical states. Other examples include fluid dynamics model for water system [30] and electromagnetics model power grid [35]. These solutions mainly monitor the behaviors of a specific physical system, and thus they can not directly detect general software exploits.

Regarding the anomaly detection based on cyber models, Yoon *et al.* [62] proposed a lightweight method for detecting anomalous executions using a distribution of system call frequencies. The authors in [9] proposed a hardware based approach for control flow graph (CFG) validation in runtime embedded systems. C-FLAT [11] is the most related work to our approach. Both C-FLAT and *e*FSA target at designing a general approach for detecting anomalous executions of embedded systems software. However, C-FLAT is insufficient to detect data-oriented attacks due to the lack of runtime execution context checking. In addition to profiling program behaviors based on control flows, several research works utilize timing information as a side channel to detect malicious intrusions[61, 65]. ContexIoT [32] provides context identification for sensitive actions in the permission granting process of IoT applications on Android platforms. Though both ContextIoT and *e*FSA consider execution contextual integrity, ContextIoT does not support the detection of data-oriented attacks.

Distinctive from existing works in this area, our *Orpheus focuses on utilizing the event-driven feature in control-program anomaly detection and our program behavior model combines both the cyber and physical aspects.* Consequently, physics-based models, which can be inherently integrated into our approach to enhance security and efficiency, do not compete but rather complement our scheme. Stuxnet attack [33] manipulated the nuclear centrifuge's rotor speed, and fooled the system operator by replaying the recorded normal data stream during the attack [28]. Since *e*FSA's detection is independent on the history data, it makes Stuxnet-like attacks detectable in *e*FSA by detecting inconsistencies between the physical context (runtime rotor speed) and the control program's behavior.

8.2 Program Behavior Modeling

Program behavior modeling has been an active research topic over the past decade and various models have been proposed for legacy applications [50]. Warrender *et al.* [59] presented the comparison of four different program behavior models, including simple enumeration of sequences, sequence frequency-based (*i.e.*, n-gram), rule induction-based data mining approach, and Hidden Markov Model (HMM). Sekar *et al.* [47] proposed to construct an FSA via dynamic learning from past traces. Recently, Xu *et al.* [60] proposed a probabilistic HMM-based control flow model representing the expected call sequences of the program for anomaly detection. Shu *et al.* [48, 49] proposed an anomaly detection approach with two-stage machine learning algorithms for large-scale program behavioral modeling. Different from these program behavior models for legacy applications, in this paper, we propose a customized *e*FSA model for detecting anomalies in embedded control programs.

It is interesting that the design paradigm of *Orpheus*, *i.e.*, augmenting physical event constraints on top of a program behavior model, *can be applied to most of the aforementioned program behavior models.* For example, HMM-based models [60] can be enhanced with event checking on event dependent state transitions. For the n-gram model [59], it is possible we identify event-dependent n-grams in the training phase and apply the event checking when observing any event-dependent n-gram in testing. In addition, control-flow integrity [10, 52] can also be augmented with event checking before executing control tasks.

Approaches	Control-oriented attacks	Data-oriented attacks	
		Control branch	Control intensity
FSA [47], HMM [60], etc.	✔	✗	✗
C-FLAT [11]	✔	✗	Limited
Our *e*FSA	✔	✔	✔

Table 3: Security guarantees of anomaly detection approaches

Table 3 summaries the security guarantees of different anomaly detection approaches. C-FLAT [11] instruments target control programs to achieve the remote attestation of execution paths of monitored programs, and the validity of control flow paths is based on static analysis. It can only partially detect control intensity attacks with the assumption of knowing legal measurements of the target program. However, if the legal measurement covers a large range of sensor values, attacks can easily evade its detection because it does not check runtime consistency between program behavior and physical context. Existing program anomaly detection models (*e.g.*, FSA [47] or HMM [60]) mainly focus on control flow integrity checking, and thus can not detect runtime data-oriented attacks. *e*FSA focuses on detecting data-oriented exploits, and the capability for detecting control-oriented exploits inherits from the FSA.

9 CONCLUSION

In this work, we presented *Orpheus*, a new security mechanism for embedded control programs in defending against data-oriented attacks, by enforcing cyber-physical execution semantics. As an FSA-based instantiation of *Orpheus*, we proposed the program behavior model *e*FSA, which advances the state-of-the-art program behavior modelling. To the best of our knowledge, this is the first anomaly detection model that integrates both cyber and physical properties. We implemented a proof-of-concept prototype to demonstrate the feasibility of our approach. Three real-world case studies demonstrated *e*FSA's efficacy against different data-oriented attacks. As for our future work, we plan to integrate physics-based models into our approach, design robust event verification mechanisms, and extend the *Orpheus* design paradigm to support actuation integrity for fine-grained anomaly detection at the instruction level without the need of tracing facilities.

ACKNOWLEDGMENTS

This work has been supported by the Office of Naval Research under Grant ONR-N00014-17-1-2498, and Security and Software Engineering Research Center (S2ERC), a NSF sponsored multi-university Industry/University Cooperative Research Center (I/UCRC).

REFERENCES

[1] Arduino. www.arduino.cc/. [Accessed 09-12-2017].
[2] Cyber-Physical Systems. www.cpse-labs.eu/cps.php. [Accessed 09-12-2017].
[3] German Steel Mill Meltdown. securityintelligence.com/german-steel-mill-meltdown-rising-stakes-in-the-internet-of-things/. [Accessed 09-12-2017].
[4] GPIO access library for RPI. wiringpi.com/. [Accessed 09-12-2017].
[5] LLVM. http://llvm.org/. [Accessed 09-12-2017].
[6] Opto 22 connects real-world industrial devices to millions of Raspberry Pi. www.prweb.com/releases/2016/11/prweb13853953.htm. [Accessed 09-12-2017].
[7] The REX Control System for Raspberry Pi. www.rexcontrols.com/. [Accessed 09-12-2017].

[8] ZingBox: Enabling the Internet of Trusted Things. www.zingbox.com/. [Accessed 09-12-2017].

[9] F. A. T. Abad, J. V. D. Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan. On-chip control flow integrity check for real time embedded systems. In *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications*, 2013.

[10] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS*, 2005.

[11] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: control-flow attestation for embedded systems software. In *CCS*, 2016.

[12] S. Adepu, S. Shrivastava, and A. Mathur. Argus: An orthogonal defense framework to protect public infrastructure against cyber-physical attacks. *IEEE Internet Computing*, 20(5):38–45, 2016.

[13] Verge Adrien, Ezzati-Jivan Naser, and Dagenais Michel R. Hardware-assisted software event tracing. *Concurrency and Computation: Practice and Experience*, 2017.

[14] H. Almohri, L. Cheng, D. Yao, and H. Alemzadeh. On threat modeling and mitigation of medical cyber-physical systems. In *IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 114–119, 2017.

[15] M. Bahrepour, N. Meratnia, and P. J. M. Havinga. Sensor fusion-based event detection in wireless sensor networks. In *Annual International Mobile and Ubiquitous Systems: Networking Services, MobiQuitous*, 2009.

[16] Alvaro A. Cárdenas, Saurabh Amin, Zong-Syun Lin, Yu-Lun Huang, Chi-Yen Huang, and Shankar Sastry. Attacks against process control systems: Risk assessment, detection, and response. In *ASIACCS*, 2011.

[17] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[18] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards automated dynamic analysis for Linux-based embedded firmware. In *NDSS*, 2016.

[19] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security*, 2005.

[20] Kyong-Tak Cho, Kang G. Shin, and Taejoon Park. CPS approach to checking norm operation of a brake-by-wire system. In *ICCPS*, 2015.

[21] Mutz Darren, Valeur Fredrik, Vigna Giovanni, and Kruegel Christopher. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, 2006.

[22] Patricia Derler, Edward A. Lee, Stavros Tripakis, and Martin Törngren. Cyber-physical system design contracts. In *ICCPS*, 2013.

[23] H.H. Feng, J.T. Giffin, Yong Huang, S. Jha, Wenke Lee, and B.P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE S&P*, 2004.

[24] Aurélien Francillon and Claude Castelluccia. Code injection attacks on Harvard-architecture devices. In *CCS*, 2008.

[25] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *SecuCode*, 2009.

[26] Debin Gao, Michael K. Reiter, and Dawn Song. On gray-box program tracking for anomaly detection. In *USENIX Security*, 2004.

[27] Luis Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A. Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *NDSS*, 2017.

[28] Luis Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A. Zonouz. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. In *NDSS*, 2017.

[29] J. Habibi, A. Gupta, S. Carlsony, A. Panicker, and E. Bertino. MAVR: Code reuse stealthy attacks and mitigation on unmanned aerial vehicles. In *ICDCS*, pages 642–652, 2015.

[30] Dina Hadžiosmanović, Robin Sommer, Emmanuele Zambon, and Pieter H. Hartel. Through the eye of the PLC: Semantic security monitoring for industrial processes. In *ACSAC*, 2014.

[31] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE S&P*, 2016.

[32] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *NDSS*, 2017.

[33] David Kushner. The real story of stuxnet. *IEEE Spectrum*, 50(3):48 – 53, 2013.

[34] Lap Chung Lam and Chiueh Tzi-cker. Automatic extraction of accurate application-specific sandboxing policy. In *RAID*, 2004.

[35] Yao Liu, Peng Ning, and Michael K. Reiter. False data injection attacks against state estimation in electric power grids. In *CCS*, 2009.

[36] Sixing Lu and Roman Lysecky. Analysis of control flow events for timing-based runtime anomaly detection. In *Proceedings of Workshop on Embedded Systems Security*, 2015.

[37] Stephen McLaughlin, Devin Pohly, Patrick McDaniel, and Saman Zonouz. A trusted safety verifier for process controller code. In *NDSS*, 2014.

[38] R. Mitchell and I.-R. Chen. Behavior rule specification-based intrusion detection for safety critical medical cyber physical systems. *IEEE Transactions on Dependable and Secure Computing*, 12(1):16–30, 2015.

[39] Robert Mitchell and Ing-Ray Chen. Adaptive intrusion detection of malicious unmanned air vehicles using behavior rule specifications. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(5):593–604, 2014.

[40] Robert Mitchell and Ing-Ray Chen. A survey of intrusion detection techniques for cyber-physical systems. *ACM Comput. Surv.*, 46(4):55:1–55:29, March 2014.

[41] Zhenyu Ning and Fengwei Zhang. Ninja: Towards transparent tracing and debugging on arm. In *USENIX Security*, 2017.

[42] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, and Ahmad-Reza Sadeghi. Hardscope: Thwarting DOP with hardware-assisted run-time scope enforcement. *CoRR*, abs/1705.10295, 2017.

[43] N. M. Rodday, R. d. O. Schmidt, and A. Pras. Exploring security vulnerabilities of unmanned aerial vehicles. In *NOMS*, 2016.

[44] Ishtiaq Rouf, Rob Miller, Hossen Mustafa, Travis Taylor, Sangho Oh, Wenyuan Xu, Marco Gruteser, Wade Trappe, and Ivan Seskar. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *USENIX Security*, 2010.

[45] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE S&P*, 2010.

[46] M. Schwartz, J. Mulder, A. R. Chavez, and B. A. Allan. Emerging techniques for field device security. *IEEE Security and Privacy*, 12(6):24–31, 2014.

[47] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE S&P*, 2001.

[48] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *CCS*, 2015.

[49] Xiaokui Shu, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Trent Jaeger. Long-span program behavior modeling and attack detection. *ACM Transactions on Privacy and Security*, 20(4):1–28, September 2017.

[50] Xiaokui Shu, Danfeng (Daphne) Yao, and Barbara G. Ryder. A formal framework for program anomaly detection. In *RAID*, 2015.

[51] S. Sridhar, A. Hahn, and M. Govindarasu. Cyber-physical system security for the electric power grid. *Proceedings of the IEEE*, 100(1):210–224, 2012.

[52] J. Tan, H. J. Tay, U. Drolia, R. Gandhi, and P. Narasimhan. PCFIRE: Towards provable preventative control-flow integrity enforcement for realistic embedded software. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016.

[53] R. Tan, H. H. Nguyen, E. Y. S. Foo, X. Dong, D. K. Y. Yau, Z. Kalbarczyk, R. K. Iyer, and H. B. Gooi. Optimal false data injection attack against automatic generation control in power grids. In *ICCPS*, 2016.

[54] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L. Littman. Practical trigger-action programming in the smart home. In *CHI*, 2014.

[55] David I. Urbina, Jairo A. Giraldo, Alvaro A. Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. Limiting the impact of stealthy attacks on industrial control systems. In *CCS*, 2016.

[56] Junia Valente, Carlos Barreto, and Alvaro A. Cárdenas. Cyber-physical systems attestation. In *DCOSS*, 2014.

[57] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE S&P*, 2001.

[58] Yong Wang, Zhaoyan Xu, Jialong Zhang, Lei Xu, Haopei Wang, and Guofei Gu. Srid: State relation based intrusion detection for false data injection attacks in scada. In *ESORICS*, 2014.

[59] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *IEEE S&P*, 1999.

[60] Kui Xu, D.D. Yao, B.G. Ryder, and Ke Tian. Probabilistic program modeling for high-precision anomaly classification. In *CSF*, 2015.

[61] Man-Ki Yoon, S. Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. Securecore: A multicore-based intrusion detection architecture for real-time embedded systems. In *RTAS*, 2013.

[62] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. Learning execution contexts from system call distribution for anomaly detection in smart embedded system. In *IoTDI*, 2017.

[63] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *ICSE '03*, 2003.

[64] Yanyan Zhuang, Eleni Gessiou, Steven Portzer, Fraida Fund, Monzur Muhammad, Ivan Beschastnikh, and Justin Cappos. Netcheck: Network diagnoses from blackbox traces. In *USENIX NSDI*, pages 115–128, 2014.

[65] Christopher Zimmer, Balasubramanya Bhat, Frank Mueller, and Sibin Mohan. Time-based intrusion detection in cyber-physical systems. In *ICCPS*, 2010.