Checking is Believing: Event-Aware Program Anomaly Detection in Cyber-Physical Systems

Long Cheng, *Member, IEEE,* Ke Tian, Danfeng (Daphne) Yao, *Senior Member, IEEE,* Lui Sha, *Fellow, IEEE,* and Raheem A. Beya, *Senior Member, IEEE*

Abstract—Securing cyber-physical systems (CPS) against malicious attacks is of paramount importance because these attacks may cause irreparable damages to physical systems. Recent studies have revealed that control programs running on CPS 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 new security methodology for defending against data-oriented attacks by enforcing cyber-physical execution semantics. We first present a general method for reasoning cyber-physical execution semantics of a control program (*i.e.*, causal dependencies between the physical context and program control flows), including the event identification and dependence analysis. As an instantiation of *Orpheus*, we then 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 a specific physical eventis missing along with the corresponding event dependent state transition. 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$\sim$0.211s for the cyber-physical contextual consistency checking.

Index Terms—Cyber-physical systems, Data-oriented attacks, Program anomaly detection, Cyber-physical execution semantics.

1 Introduction

Cyber-physical systems (CPS) consist of a tightly coupled integration of computational elements and physical components. The computational elements rely on sensors to monitor the physical environment and make control decisions to affect physical processes with feedback loops [3]. These systems are widely used to operate critical infrastructure assets, such as electric power grid, oil and natural gas distribution, industry automation, medical devices, automobile systems, and air traffic control [4]. In the industrial control domain, CPSs are instantiated as the Industrial Control Systems (ICS), Distributed Control Systems (DCS), or Supervisory Control and Data Acquisition (SCADA) systems [5]. Though CPS and IoT (Internet of Things) 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 [6].

The tight coupling with physical space of CPS brings new security and safety challenges. Control programs running on CPS devices monitor physical environments by taking sensory data as input and send control signals that affect physical environments or processes [7]. They are critical to the proper operations of CPS, as anomalous program behaviors can have serious consequence, or even cause devastating damages to physical systems [8]. For example, the Stuxnet [9] attack allows hackers to compromise the control system of a nuclear power plant and manipulate real-world equipment such as centrifuge rotor speeds, which can be very dangerous. According to ICS-CERT's report [10], there have been continuously increasing number of cyber attacks targeting critical infrastructure. Therefore, securing CPS against malicious attacks becomes of paramount importance in the prevention of potential damages to physical systems.

Recent studies [8], [11], [12], [13], [14], [15] 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 [16] or code reuse attacks [12]. Control-oriented attacks in conventional cyber systems (*i.e.*, without cyber-physical interactions) have been well studied [17]. It is possible that existing detection approaches [18], [19], [20], [21], [22] are extended to defend against control-oriented attacks in embedded systems software.
- *Data-oriented attacks* manipulate program's internal data variables without violating its control-flow integrity (CFI), *e.g.*, non-control data attacks [23], control-flow bending [22], data-oriented programming [24]. 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. We mainly focus on runtime software exploits, and thus sensor data spoofing attacks [25], [26] in the physical domain are out of the scope in this work.

Since many control decisions are made based on particular

- *Long Cheng, Ke Tian and Danfeng (Daphne) Yao are with the Department of Computer Science, Virginia Tech, VA, USA.*
- *Lui Sha is with the Department of Computer Science, University of Illinois at Urbana Champaign, Champaign, IL, USA.*
- *Raheem A. Beya is with the School of Electrical and Computer Engineering, Georgia Tech, GA, USA.*

A preliminary version of this work appeared in [1] and [2].

values of data variables in control programs [8], 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 to flow into a tank despite it is full [27] or preventing a blast furnace from being shut down properly as in the recent German steel mill attack [28]). *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 [8]).

In many instances, CPS can be modeled as *event-driven* control systems [29], [30]. 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. Defending against CPS data-oriented attacks is challenging due to the following reasons. First, data-oriented exploits can achieve attack goals without incurring illegal control flows, thus providing opportunities for attackers to evade all control flow integrity based detections [24]. Second, CPS programs normally rely on external sensor events to make control decisions. This physical event-driven nature makes it difficult to predict runtime program behaviors in CPS. Hence, an anomaly detection system needs to check the *runtime integrity* of program behaviors from *both cyber and physical domains*. Unfortunately, there exist very few defenses [8], [31] and they are ineffective to prevent both attack types due to the lack of runtime execution semantics checking.

Goals and Contributions. In this paper, we focus on a new type of runtime attacks that result in inconsistencies between the physical context and program execution, where executed control flow paths do not correspond to the observed events. These attacks do not necessarily violate any control flow integrity, so existing techniques based on control flow checking are not effective. 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 [27]. None of existing program anomaly detection solutions [17] 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 [32] or HMM model [33].
- We implement a proof-of-concept prototype on Raspberry Pi platforms, which have emerged as popular devices for building CPS applications [8], [34], [35]. 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 real-world CPS 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 $\sim$0.0001s to check each state transition in *e*FSA model, $\sim$0.063s for the local event verification, and $\sim$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 in CPS. Our design is a general approach for event-driven embedded control systems. In Sec. 8, 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 introduce the CPS background, and describe the attack model of this work. We use examples to illustrate our new detection capabilities, and then present the design overview of *Orpheus* framework.

2.1 CPS Background

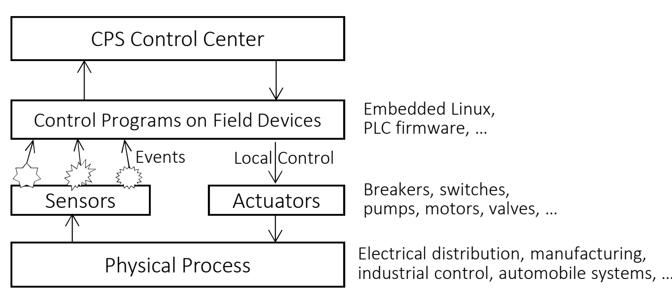


Fig. 1: An abstract view of the event-driven CPS architecture. CPS is exposed with a large attack surface and attacks can be launched across all components in the system. Existing CPS anomaly detection approaches mainly monitor behaviors of the physical process. On the contrary, we focus on anomaly detection for CPS programs running on field devices or the central control center.

Fig. 1 shows an abstract view of the CPS system architecture, which is also in line with the architecture of modern Industrial Control Systems (ICS). 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 [36]. 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. CPS 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.

Embedded devices (*a.k.a.* field devices) in CPS are situated in the field, where their operating systems are typically embedded Linux/Windows variants [37] or PLC firmware [36]. Traditionally, 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 CPS control programs increasingly vulnerable [37].

2.2 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 CPS field 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 [21]. 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 [31]. 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 [8]. 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 [11].
- *Our focus.* We focus our investigation on runtime software exploits, and thus sensor data spoofing attacks in the physical domain [26] 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 [38]. Despite system call based monitoring is widely used for detecting compromised programs, we aim at developing a CPS-specific anomaly detection system by augmenting an existing program behavior model with physical context awareness.

2.3 New Detection Capabilities

Our new detection capability is detecting data-oriented attacks in CPS 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 [1]. 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. 2. 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. 2(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. 2(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.

```
1  while(…){
2    eventRead();
3    if(Push_Event())
4      push-syringe();
5    else if(Pull_Event())
6      pull-syringe();
7    …
8  }
```

(a)

```
9  push-syringe(){
10   steps = … ;
11   for(i=0;i<steps;i++)
12   {
13     write(i2c,…);
14     …
15   }
16 }
```

(b)

Fig. 2: Two examples of data-oriented software exploits in a real-world CPS 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.

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 [8], which is based on the attestation of control flows and a finite number of permitted execution patterns, cannot fully detect these attacks. Similarly, recent frequency- and co-occurrence-based anomaly detection approaches (*e.g.*, global trace analysis [39] and system call frequency distribution (SCFD) [31]) cannot detect such either type of attacks, as their analyses do not model runtime cyber-physical context dependencies.

2.4 Definition of Events

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

1. https://hackaday.io/project/1838-open-syringe-pump

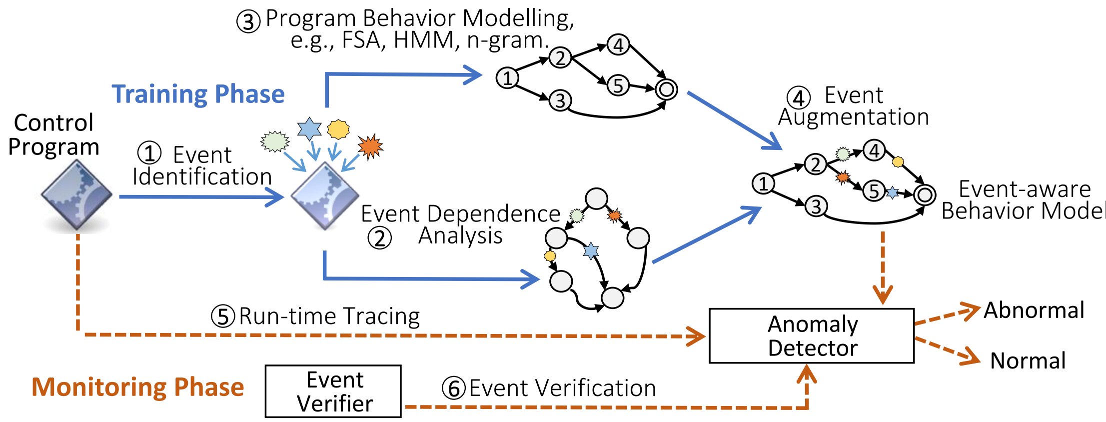


Fig. 3: Workflow of *Orpheus* event-aware anomaly detection framework, which augments an existing program behavior model with cyber-physical contextual integrity.

- 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. 2). Such events are commonly pre-defined and used in CPS/IoT's trigger-action programming ("if, then") model [30], [40].
- 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. 2. It is challenging to identify non-binary events since they are not explicitly declared in control programs.

2.5 *Orpheus* Design Overview

Fig. 3 shows the workflow of *Orpheus* event-aware anomaly detection framework, which is 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). In particular, to capture the cyber-physical context dependency of control programs, the training stage in *Orpheus* encompasses both static program analysis and dynamic profiling.

There are four main steps in the training phase. In step ①, *Orpheus* identifies both binary events and non-binary events involved in the control program. In step ②, it performs the program dependency analysis to generate event-annotated CFG, which identifies the instructions/statements associated with binary events, and control intensity loops associated with non-binary events. In step ③, *Orpheus* conducts the program behavior modelling, such as the HMM-based model [33], n-gram model [32], or control-flow integrity [18], which we refer to as a basic program behavior model in *Orpheus*. The next step ④ is important. It augments the basic model with event constraints and obtains the event-aware program behavior model.

Steps ⑤ and ⑥ are the testing phase. In step ⑤, *Orpheus* monitors the CPS control program's execution and collects runtime traces. The basic program behavior model normally aims at detecting control-oriented attacks. Our main contribution lies in the event awareness enhancement on top of a basic model. Whenever an event-dependent control-flow path is encountered in step ⑤, the event verifier checks the consistency between runtime behavior and program execution semantics, *i.e.*, whether a specific physical event associated with this event-dependent state transition is observed in the physical domain. 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 Reasoning About Cyber-Physical Execution Semantics

In this section, we present a general method for reasoning about cyber-physical execution semantics of a control program through static analysis, including the event identification and dependence analysis.

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 [41] 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[2] in our static analysis.

According to the definition of a non-binary event in Sec. 2.1, 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. Algorithm 1 describes our static analysis for identifying non-binary events. We first obtain the LLVM Intermediate Representation (IR) of a control program P using the `Clang` compiler [41], and construct the program dependence graph (PDG), including both data and control dependencies (Line 4). The control dependence graph is at the basic block level[3], while the data dependency graph is at the granularity of instructions. Then, we obtain all conditional branch instructions with loops, by searching

2. 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 [42].

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

the conditional "br" instruction, which takes a single "i1" value and two "label" values (Line 5). For each conditional branch with a loop, we conduct the backward inter-procedural dataflow analysis to find any prior data dependence on sensor-reading APIs (Line 7). Then, we 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 [43] (Line 9). If a loop statement is data-dependent on external sensor data, and triggers a certain control action, we identify a non-binary event (Line 11). In each iteration, we record the identified non-binary event and control intensity loop (Line 12), which is the output of the event identification process.

Algorithm 1: Identifying non-binary events

1 **Input:** Program P; Sensor-reading API set API_{sens}; Actuation API set API_{actu}
2 **Output:** Non-binary-event set E_{nb}
3 $E_{nb} \leftarrow \emptyset$;
4 G_{pdg} = `ConstructPDG(`P`)` */*construct the program dependence graph*/;*
5 LoopBrSet = `getLoopBrSet(`P`)` */*get all the conditional branch instructions with loops*/;*
6 **for** *BrInst*=`getNextInst(`*LoopBrSet*`)` **do**
7   S_{bdd} = `BackwardDataDependence(`G_{pdg}, *BrInst*`)`;
8   */*Backward data dependent statements on $BrInst$*/;*
9   S_{fcd} = `ForwardContDependence(`G_{pdg}, *BrInst*`)`;
10   */*Forward control dependent statements on $BrInst$*/;*
11   **if** *($S_{bdd} \cap API_{sens} \neq \emptyset$ and $S_{fcd} \cap API_{actu} \neq \emptyset$)* **then**
12   E_{nb}= $E_{nb} \cup$ `Event(`*BrInst*,S_{bdd},S_{fcd}`)`;
13 **end**

A more specific example of our event identification is illustrated in Fig. 4 using a C-based control program as an example. The figure shows a non-binary event represented by LLVM IR after the data dependence and control dependence analysis (❶). We then locate a conditional branch instruction with a loop (❷). Suppose this conditional branch is data dependent on a sensor-reading API (❸). On its true branch, if we find any actuation API (❹), we consider the loop as a non-binary event. Finally, we record the search results for the next event dependence analysis (❺).

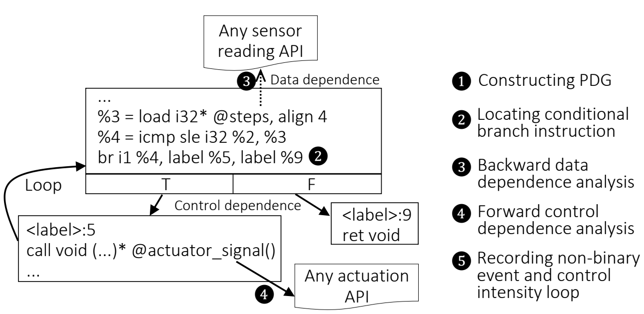


Fig. 4: An example of identifying non-binary events

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 `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.

Algorithm 2 describes our event dependence analysis for nested binary events. Given a binary-event triggered basic block BB_{eta}, we backward traverse all its control dependent blocks until reaching the root in a recursive manner, and extract corresponding branch labels (*i.e.*, `True` or `False`). In the recursive function `FindEveDependence` (Line 5), once we find a basic block on which BB_{cur} is control dependent (Line 7), we check whether it contains any external event (Line 9). If yes, we add this event together with its branch label to E_b (Line 10). The condition $E_b \cap E_{tmp} = \emptyset$ avoids potential loops when including new events into E_b. Then, we recursively search any upstream event that BB_{cur} depends on (Line 12).

Algorithm 2: Event dependence analysis for binary events

1 **Input:** Event-triggered basic block BB_{eta}; Control flow graph G_{cfg} of program P;
2 **Output:** E_b: events that trigger the execution of BB_{eta}
3 $E_b \leftarrow \emptyset$;
4 BB_{cur} = BB_{eta};
5 **Function** `FindEveDependence` *(BB_{cur}, G_{cfg}, S_{eb})*
6   **for** BB_{tmp}= `getNextBB(`G_{cfg}`)` **do**
7   **if** *($BB_{tmp}.toid$ == BB_{cur})* **then**
8   E_{tmp}=`GetEvent(`BB_{tmp}`)`;
9   **if** $E_{tmp} \neq \emptyset$ *and* $E_b \cap E_{tmp} = \emptyset$ **then**
10   E_b= $E_b \cup E_{tmp}$;
11                 $BB_{cur} = BB_{tmp}$;
12   `FindEveDependence` (BB_{cur}, G_{cfg}, E_b);
13   **end**
14   return;

Fig. 5 illustrates an example of our event dependence analysis. Block 18 (*i.e.*, the label id) 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. We finally identify three event-dependent basic blocks, and obtain the corresponding event-annotated CFG.

In addition to the static analysis approach, an alternative for event dependence analysis is using dynamic slicing [44], 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.

4 *e*FSA: An Instantiation of *Orpheus*

In this section, we describe details about how to build the event-aware finite-state automaton (*i.e.*, *e*FSA) model, a system call level

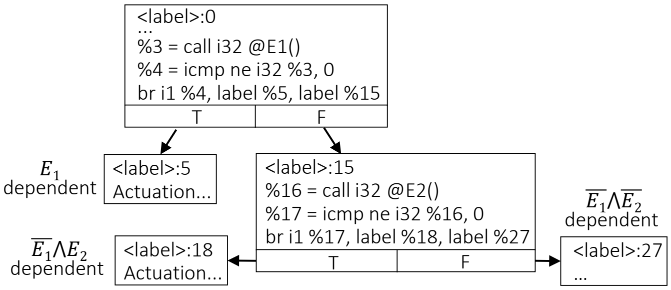


Fig. 5: Event dependence analysis for nested events

FSA-based instantiation of the *Orpheus* framework. *e*FSA captures the event-driven feature of CPS programs to detect evasive attacks.

4.1 Formal Description of *e*FSA

We construct the finite-state automaton (FSA) [45] 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 [17], [46].

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,. . . ,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.

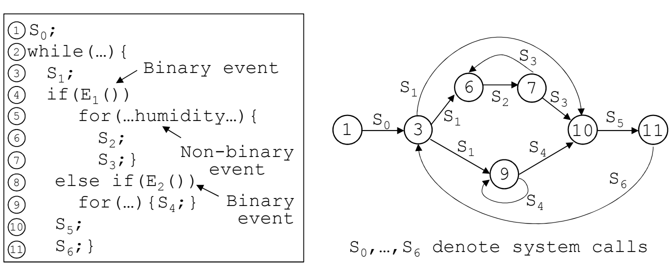


Fig. 6: System call based finite-state automaton (FSA) model: (a) an example program; (b) the corresponding FSA model.

Our *e*FSA model extends FSA with external context constraints, where event-dependent state transitions in FSA are labeled with event constraints. 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.

4.2 From Event-Annotated CFG to *e*FSA

To construct an *e*FSA model, we need to identify event-dependent state transitions at the system call level in FSA. Towards this end, we apply the event dependence analysis results (described in Sec. 3.1 and 3.2) to transform instruction-level dependencies in LLVM IR 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 [47], [48]. In *e*FSA, we adopt a dynamic profiling based approach to discover event dependent state transitions. We first transform instruction-level event dependencies in LLVM IR to statement-level dependencies in source code with line numbers. Then, we map line numbers and file names to return addresses (*e.g.*, by using the `addr2line` tool) that are collected in the dynamic profiling phase when the FSA model is constructed. In this way, we obtain the system call level event-dependent state transitions in FSA. Subsequently, we augment the event-driven information over the underlying FSA, and finally construct the *e*FSA model.

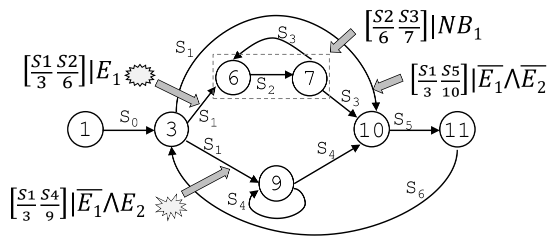


Fig. 7: An example of the *e*FSA model

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. Through the event dependence analysis, we identify that lines 5-7 (where S_2 and S_3 are invoked) and line 9 (where S_4 is invoked) are dependent on the binary events E_1 and E_2, respectively. To avoid redundancy, we associate a binary event to the first state transition in FSA that is dependent on it. For a non-binary event, we associate it with the control intensity loop. In Fig. 7, we identify binary-event dependent state transitions $[\frac{S_1}{3}\frac{S_2}{6}]|E_1$, $[\frac{S_1}{3}\frac{S_4}{9}]|\overline{E_1} \wedge E_2$, 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 $[\frac{S_1}{3}\frac{S_5}{10}]|(\overline{E_1} \wedge \overline{E_2})$.

4.3 Security Policies in *e*FSA

*e*FSA expresses causal dependencies between physical events and program control flows. By checking execution semantics (*i.e.*, enforcing cyber-physical security policies) at runtime, *e*FSA improves the robustness against data-oriented attacks by increasing the difficulties that an attack could bypass the anomaly detection.

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.

4.4 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 a 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 [49] 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 [34]. 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. 7.4). 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.

4.5 Generalization of *e*FSA

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, it is also applicable to network event-triggering scenarios. For example, we consider each type of network packet as an event, and the *e*FSA model is augmented with network events. Such an *e*FSA model can detect false command injection attacks. It checks the consistency of system call traces at the receiver and sender, ensuring their system call invocations conforming to the network API semantics [50].

5 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.

5.1 Runtime Monitoring and Detection

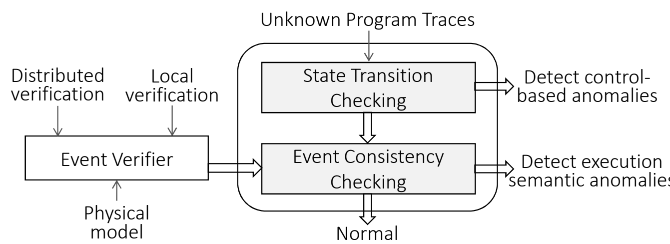


Fig. 8: An instance of detecting attacks on control branch

Our anomaly detector traces system calls as well as the corresponding PC values during the execution of a control program. As shown in Fig. 8, the anomaly detection is composed of an event verifier and two checking 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 [16]) which violate control flow integrity of the model.
- **Event-dependent state transition.** In case of an event dependent state transition according to the *e*FSA model, we first perform the above basic state-transition checking. More importantly, with the help of the event verification (discussed in Sec. 5.2), 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.

5.2 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:* which is complementary to the runtime event verification. Cyber-physical inconsistency may be detected based on physical models [51]. For example, one may utilize fluid dynamics and electromagnetics as the basic laws to create prediction models for water system [52] and power grid [53]. 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.

6 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 [8], [34], [35]. 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. 7). 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]
```

Fig. 9: 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. 9 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)[4] compiler infrastructure, based on an open source static slicer[5] 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.

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`[6] and `c-sense-hat`[7]. The event library reads pressure 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.

7 Experimental Validation

We conduct CPS case studies, 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. 7.2)?
- Whether *e*FSA is able to detect different data-oriented attacks (Sec. 7.3 and 7.4)?

4. http://llvm.org/
5. https://github.com/mchalupa/dg
6. http://experix.sourceforge.net/
7. https://github.com/davebm1/c-sense-hat

7.1 CPS Case Studies

Solard[8]**.** 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[9]**.** It was developed as an embedded application for Arduino platform. Abera *et al.* [8] 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 [8]. 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.

7.2 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.

We first measure the time taken for training models in our prototype, where the main overhead comes from the event dependence analysis. 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.

	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

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$\sim$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 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.

8. https://github.com/mrpetrov/solarmanpi
9. https://github.com/control-flow-attestation/c-flat

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

7.3 Detecting Attacks on Control Branch

In this experiment, we evaluate *e*FSA's security guarantees against control branch attacks.

7.3.1 Solard

In Solard, we engineer a buffer overflow vulnerability and manipulate 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 [28], 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$^\circ C$, it turns on the heater. And when the temperature is higher than 60$^\circ C$, 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.

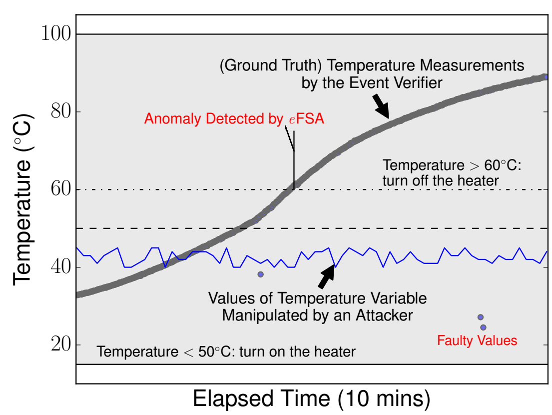


Fig. 10: An instance of Solard experiment

Fig. 10 illustrates an instance of the Solard experiment. We corrupt the temperature sensor values in the range of 40$\sim$45$^\circ 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. 10. 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.

We did encounter sensor measurement failures, *e.g.*, isolated dots as shown in Fig. 10. 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 [54] can be applied to enhance the detection accuracy.

7.3.2 SyringePump

In SyringePump, we set the threshold to $40rH$, *i.e.*, when the relative humidity value is higher than $40rH$, it drives the movement of syringe pump by sending control signals to dispense liquid. The buffer overflow attack manipulates the humidity sensor values to purposely trigger `event-push` control actions without receiving an external event or environmental trigger. Such an attack leads to unintended but valid control flows.

Fig. 11 illustrates an example of the experiment. The remote user command corrupts the humidity sensor value to be $48.56rH$, which falsifies the return value of `event-push` to be `True`. 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. In case of any event-driven state transition according to *e*FSA, the event verifier checks consistency between the runtime execution semantics (*e.g.*, the instantaneous humidity value) and program internal state. As shown in Fig. 10, *e*FSA raises an alarm when it finds a mismatch between the execution semantics and program behavior.

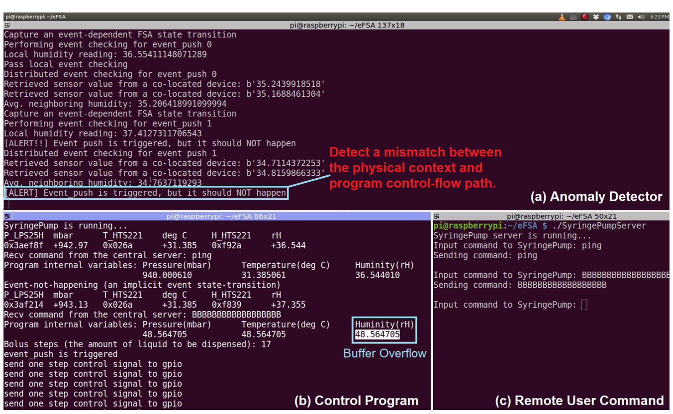
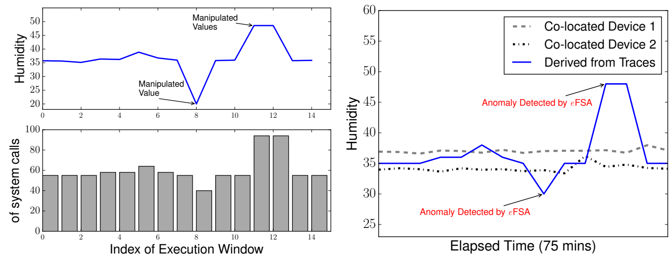


Fig. 11: An instance of SyringePump experiment

7.4 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 $30rH$. The corrupted humidity value determines the amount of liquid to be dispensed, which equals to the humidity value subtracted by $30rH$ in this test. In the training stage, we 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 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

Fig. 12: An instance of SyringePump experiment with a sampling rate of 5 minutes

Fig. 12(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. 12(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 7.3 and Sec. 7.4, 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.

8 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 [55]), 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 [8], [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.

Bare-metal CPS Devices: 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 [35], [37]. For example, Raspberry Pi devices with embedded Linux OS have been used as field devices in many CPS/IoT applications [57]. Linux-based PLCs for industrial control have emerged to replace traditional PLCs [58] for deterministic logic execution. However, embedded devices may still operate in bare-metal mode [8], 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 [59]. 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 [39]. 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 [60]. The recent work, Ninja [61], 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.

9 Related Work

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

9.1 Anomaly Detection in CPS

Due to the diversity of CPS applications, existing anomaly detection solutions are proposed to detect specific attacks for specific applications, such as smart infrastructures [4], unmanned aerial vehicles [62], medical devices [63], automotive [64], [65], industrial control process [5], [34], [51]. The majority of research efforts in this area thus far have concentrated on behavior model-based anomaly detection [51], and can be generally classified into two categories: *1)* cyber model (*e.g.*, program behavior model, network traffic analysis, or timing analysis); *2)* physical model (*e.g.*, range-based model or physical laws). Our proposed *e*FSA analyzes both the cyber and physical properties of CPS, as well as their interactions. Thus, we refer to it as the cyber-physical model. Table 3 compares representative CPS anomaly detection solutions.

- *Program behavior model.* Regarding the CPS anomaly detection based on program behavior models in the cyber domain, Yoon *et al.* [31] proposed a lightweight method for detecting anomalous executions using the distribution of system call frequencies. The frequencies are for individual system calls, *i.e.*, 1-grams. The authors in [20] proposed a hardware based approach for control-flow graph (CFG) validation in runtime embedded systems. McLaughlin *et al.* [34] presented the Trusted Safety Verifier (TSV) to verify safety-critical code executed on programmable controllers, such as checking safety properties like range violations and interlocks of PLC programs. C-FLAT [8] 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. Given an aggregated authenticator (*i.e.*, fingerprint) of the program's control flow computed by the prover, the verifier is able to trace the exact execution path and thus can determine whether application's control flow has been compromised. C-FLAT [8] 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. 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.
- *Traffic-based model.* Control systems exhibit relatively simpler network dynamics compared with traditional IT systems, *e.g.*, fixed network topology, regular communication patterns, and a limited number of communication protocols. As a result, implementing network-based anomaly detection systems would be easier than traditional mechanisms. Feng *et al.* [66] presented an anomaly detection method for ICS by taking advantage of the predictable and regular nature of communication patterns that exist between field devices in ICS networks. In the training phase, a base-line signature database for general packages is constructed. In the monitoring phase, the authors utilize Long Short-Term Memory (LSTM) network based softmax classifier to predict the most likely package signatures that are likely to occur given previously seen package traffic. The anomaly detector captures traffic anomalies if a package's signature is not within the predicted top k most probable signatures according to the LSTM-based model.
- *Timing-based model.* Several studies utilized timing information as a side channel to detect malicious intrusions. The rationale is that execution timing information is considered an important constraint for real-time CPS applications, and mimicking timing is more difficult than mimicking the execution sequence. To this end, Zimmer *et al.* [11] used the worst-case execution

Research Work	Category	Approach	Security Guarantee	Validation
Yoon *et al.* [31]	Program behavior model (cyber)	Syscall frequencies	Frequency-based program control flow anomaly	Raspberry Pi testbed
Feng *et al.* [66]	Network traffic analysis (cyber)	Machine learning based traffic analysis	Traffic alteration	Traffic data from a gas pipeline system
Zimmer *et al.* [11]	Timing analysis model (cyber)	Static/dynamic timing analysis	Code injection attacks	Simulation/Testbed
C-FLAT [8]	Program behavior model (cyber)	Program analysis and instrumentation	Control-oriented attacks and limited non-control-data attacks	Raspberry Pi testbed
Hadziosmanovic *et al.* [52]	Range-based model (physical)	Attribute values extracted from network traffic	False data injection attacks	Traffic data from water treatment plants
Cardenas *et al.* [5]	Physical laws	Linear model derived from training data	False data injection attacks	Simulation
SRID [25]	Physical laws	Correlation analysis of system variables.	False data injection attacks	Simulation
C^2 [67]	Control policies (physical)	User specified control policies	Control signal violation	Raspberry Pi testbed
*e*FSA (Our work)	Cyber-physical model	Event-aware FSA	Data-oriented attacks	Raspberry Pi testbed

TABLE 3: Comparison of representative CPS anomaly detection approaches

time (WCET) obtained through static analysis to detect code injection attacks in CPS. Such timing-based detection technique is realized by instrumenting checkpoints within real-time applications. Sibin *et al.* [68] focused on detecting intrusions in real-time control systems. Yoon *et al.* [69] presented SecureCore, a multicore architecture using the timing distribution property of each code block to detect malicious activities in real-time embedded system. Lu *et al.* [21] investigated how to reduce timing checkpoints without sacrificing detection accuracy in embedded systems.

- *Range based model.* Enforcing data ranges is the simplest method to detect CPS anomalies in the physical domain. As long as sensor readings are outside a pre-specified normal range, the anomaly detector raises an alarm. Hadziosmanovic *et al.* [52] presented a non-obtrusive security monitoring system by deriving models for PLC variables from network packets as the basis for assessing CPS behaviors. For constant and attribute series, the proposed detection approach raises an alert if a value reaches outside of the enumeration set. However, range-based detection suffers from a low detection rate because it neglects the program's execution context, *e.g.*, if the legal measurement covers a large range of sensor values, attacks can easily evade its detection.
- *Physical laws.* The idea of using physical models to define normal operations for anomaly detection is that, system states must follow immutable laws of physics. Wang *et al.* [25] derived a graph model to defeat false data injection attacks in SCADA system. It captures internal relations among system variables and physical states. Cho *et al.* [64] presented a brake anomaly detection system, which compares the brake data with the norm model to detect any vehicle misbehavior (*e.g.*, due to software bugs or hardware glitches) in the Brake-by-Wire system. Other examples include utilizing fluid dynamics and electromagnetics as the basic laws to create prediction models for water system [52] and power grid [53], respectively. Based on the prediction models and predefined threat constraints, these methods check whether sensor readings are consistent with the expected behaviors of a control system. Cardenas *et al.* [5] proposed a physical model based detection method by monitoring the physical system under control, and the sensor and actuator values. The authors also proposed automatic response mechanisms by estimating the system states. Urbina *et al.* [51] discussed the limitations of existing physics-based attack detection approaches, *i.e.*, they cannot limit the impact of stealthy attacks. The authors proposed a metric to measure the impact of stealthy attacks and to study the effectiveness of physics-based detection.
- *Control policies.* Physical model can also be specified by control policies. The main purpose of the policies is to improve the survivability of control systems, *i.e.*, without losing critical functions under attacks. For example, McLaughlin *et al.* [67] introduced a policy enforcement for governing the usage of CPS devices, which checks whether the policy allows an operation depending on the state of the plant around the time the operation was issued. The policies specify what behaviors should be allowed to ensure the safety of physical machinery and assets.
- *Cyber-physical model.* Such a model captures the cyber-physical context dependency of control programs. Our proposed *e*FSA characterizes control-program behaviors with respect to events, and enforces the runtime consistency among control decisions, values of data variables in control programs, and the physical environments. Thus, it is able to detect inconsistencies between the physical context and program execution.

As shown in Table 3, cyber models and physical models have different security guarantees. The former targets at detecting CPS control program anomalies in the cyber domain. While the latter mainly focuses on detecting false data injection attacks in the physical domain [53]. The cyber-physical interaction (*i.e.*, interactions between cyber components and physical components) in CPS makes it challenging to predict runtime program behaviors through static analysis of the program code or model training. Existing cyber models [8], [31] are effective against control-oriented attacks, however, insufficient to detect data-oriented attacks. An effective CPS program anomaly detection needs to reason about program behaviors with respect to cyber-physical interactions, *e.g.*, the decision of opening a valve has to be made based on the current water level of the tank. ContexIoT [30] 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 [9] manipulated the nuclear centrifuge's rotor speed, and fooled the system operator by replaying the recorded normal data stream during the attack [36]. Since *e*FSA's detection is independent on the history data, it makes Stuxnet-like attacks detectable in *e*FSA by detecting runtime inconsistencies between the physical context (runtime rotor speed) and the control program's behavior. In addition, attackers may exploit hardware vulnerabilities [70] to manipulate data in memory so as to launch attacks on control branch or control intensity. *e*FSA is also able to detect anomalies caused by such hardware attacks.

9.2 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 [17]. Warrender *et al.* [32] 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.* [45] proposed to construct an FSA via dynamic learning from past traces. Recently, Xu *et al.* [33] proposed a probabilistic HMM-based control flow model representing the expected call sequences of the program for anomaly detection. Shu *et al.* [39], [71] 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 CPS. Existing program anomaly detection models 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 underlying FSA.

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 [33] can be enhanced with event checking on event-dependent state transitions. For the n-gram model [32], 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 [18], [35] can also be augmented with event checking before executing control tasks.

10 Conclusion

In this work, we presented *Orpheus*, a new security mechanism for CPS 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 in part 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] L. Cheng, K. Tain, and D. D. Yao, "Enforcing cyber-physical execution semantics to defend against data-oriented attacks," in *ACSAC*, 2017.

[2] D. D. Yao, X. Shu, L. Cheng, and S. J. Stolfo, "Anomaly detection as a service: Challenges, advances, and opportunities," *Synthesis Lectures on Information Security, Privacy, and Trust*, vol. 9, no. 3, pp. 1–173, 2017.

[3] A. B. Sharma, F. Ivančić, A. Niculescu-Mizil, H. Chen, and G. Jiang, "Modeling and analytics for cyber-physical systems in the age of big data," *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 4, pp. 74–77, 2014.

[4] S. Sridhar, A. Hahn, and M. Govindarasu, "Cyber-physical system security for the electric power grid," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 210–224, 2012.

[5] A. A. Cárdenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry, "Attacks against process control systems: Risk assessment, detection, and response," in *ASIACCS*, 2011.

[6] "Cyber-Physical Systems," www.cpse-labs.eu/cps.php, [Accessed 09-12-2017].

[7] R. Mitchell and I.-R. Chen, "A survey of intrusion detection techniques for cyber-physical systems," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 55:1–55:29, Mar. 2014.

[8] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik, "C-FLAT: control-flow attestation for embedded systems software," in *CCS*, 2016.

[9] D. Kushner, "The real story of Stuxnet," *IEEE Spectrum*, vol. 50, no. 3, pp. 48 – 53, 2013.

[10] "The Industrial Control Systems Cyber Emergency Response Team," https://ics-cert.us-cert.gov/, [Accessed 09-12-2017].

[11] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan, "Time-based intrusion detection in cyber-physical systems," in *ICCPS*, 2010.

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

[13] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for Linux-based embedded firmware," in *NDSS*, 2016.

[14] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. Mohammed, and S. A. Zonouz, "Hey, my malware knows physics! attacking PLCs with physical model aware rootkit," in *NDSS*, 2017.

[15] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A. Sadeghi, "Hardscope: Thwarting DOP with hardware-assisted run-time scope enforcement," *CoRR*, vol. abs/1705.10295, 2017. [Online]. Available: http://arxiv.org/abs/1705.10295

[16] A. Francillon and C. Castelluccia, "Code injection attacks on Harvard-architecture devices," in *CCS*, 2008.

[17] X. Shu, D. D. Yao, and B. G. Ryder, "A formal framework for program anomaly detection," in *RAID*, 2015.

[18] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS*, 2005.

[19] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *SecuCode*, 2009.

[20] 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.

[21] S. Lu and R. Lysecky, "Analysis of control flow events for timing-based runtime anomaly detection," in *Proceedings of Workshop on Embedded Systems Security*, 2015.

[22] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security*, 2015.

[23] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Security*, 2005.

[24] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE S&P*, 2016.

[25] Y. Wang, Z. Xu, J. Zhang, L. Xu, H. Wang, and G. Gu, "Srid: State relation based intrusion detection for false data injection attacks in scada," in *ESORICS*, 2014.

[26] 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.

[27] S. Adepu, S. Shrivastava, and A. Mathur, "Argus: An orthogonal defense framework to protect public infrastructure against cyber-physical attacks," *IEEE Internet Computing*, vol. 20, no. 5, pp. 38–45, 2016.

[28] "German Steel Mill Meltdown," securityintelligence.com/german-steel-mill-meltdown-rising-stakes-in-the-internet-of-things/, [Accessed 09-12-2017].

[29] P. Derler, E. A. Lee, S. Tripakis, and M. Törngren, “Cyber-physical system design contracts,” in *ICCPS*, 2013.

[30] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash, “ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms,” in *NDSS*, 2017.

[31] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha, “Learning execution contexts from system call distribution for anomaly detection in smart embedded system,” in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 2017, pp. 191–196.

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

[33] K. Xu, D. Yao, B. Ryder, and K. Tian, “Probabilistic program modeling for high-precision anomaly classification,” in *CSF*, 2015.

[34] S. McLaughlin, D. Pohly, P. McDaniel, and S. Zonouz, “A trusted safety verifier for process controller code,” in *NDSS*, 2014.

[35] 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)*, 2016, pp. 1–10.

[36] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. Mohammed, and S. A. Zonouz, “Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit,” in *NDSS*, 2017.

[37] M. Schwartz, J. Mulder, A. R. Chavez, and B. A. Allan, “Emerging techniques for field device security,” *IEEE Security and Privacy*, vol. 12, no. 6, pp. 24–31, 2014.

[38] M. Darren, V. Fredrik, V. Giovanni, and K. Christopher, “Anomalous system call detection,” *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 1, pp. 61–93, 2006.

[39] X. Shu, D. Yao, and N. Ramakrishnan, “Unearthing stealthy program attacks buried in extremely long execution paths,” in *CCS*, 2015.

[40] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman, “Practical trigger-action programming in the smart home,” in *CHI*, 2014.

[41] “LLVM,” http://llvm.org/, [Accessed 09-12-2017].

[42] E. J. Schwartz, T. Avgerinos, and D. 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.

[43] “GPIO access library for RPI,” wiringpi.com/, [Accessed 09-12-2017].

[44] X. Zhang, R. Gupta, and Y. Zhang, “Precise dynamic slicing algorithms,” in *ICSE ’03*, 2003.

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

[46] D. Gao, M. K. Reiter, and D. Song, “On gray-box program tracking for anomaly detection,” in *USENIX Security*, 2004.

[47] D. Wagner and D. Dean, “Intrusion detection via static analysis,” in *IEEE S&P*, 2001.

[48] L. C. Lam and C. Tzi-cker, “Automatic extraction of accurate application-specific sandboxing policy,” in *RAID*, 2004.

[49] “Arduino,” www.arduino.cc/, [Accessed 09-12-2017].

[50] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos, “Netcheck: Network diagnoses from blackbox traces,” in *USENIX NSDI*, 2014, pp. 115–128.

[51] D. I. Urbina, J. A. Giraldo, A. A. Cardenas, N. O. Tippenhauer, J. Valente, M. Faisal, J. Ruths, R. Candell, and H. Sandberg, “Limiting the impact of stealthy attacks on industrial control systems,” in *CCS*, 2016.

[52] D. Hadžiosmanović, R. Sommer, E. Zambon, and P. H. Hartel, “Through the eye of the PLC: Semantic security monitoring for industrial processes,” in *ACSAC*, 2014.

[53] Y. Liu, P. Ning, and M. K. Reiter, “False data injection attacks against state estimation in electric power grids,” in *CCS*, 2009.

[54] 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.

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

[56] J. Valente, C. Barreto, and A. A. Cárdenas, “Cyber-physical systems attestation,” in *DCOSS*, 2014.

[57] “Opto 22 connects real-world industrial devices to millions of Raspberry Pi,” www.prweb.com/releases/2016/11/prweb13853953.htm, [Accessed 09-12-2017].

[58] “The REX Control System for Raspberry Pi,” www.rexcontrols.com/, [Accessed 09-12-2017].

[59] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller, “Formalizing sensitivity in static analysis for intrusion detection,” in *IEEE S&P*, 2004.

[60] V. Adrien, E.-J. Naser, and D. M. R., “Hardware-assisted software event tracing,” *Concurrency and Computation: Practice and Experience*, 2017.

[61] Z. Ning and F. Zhang, “Ninja: Towards transparent tracing and debugging on arm,” in *USENIX Security*, 2017.

[62] R. Mitchell and I.-R. Chen, “Adaptive intrusion detection of malicious unmanned air vehicles using behavior rule specifications,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 5, pp. 593–604, 2014.

[63] 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*, vol. 12, no. 1, pp. 16–30, 2015.

[64] K.-T. Cho, K. G. Shin, and T. Park, “CPS approach to checking norm operation of a brake-by-wire system,” in *ICCPS*, 2015.

[65] I. Rouf, R. Miller, H. Mustafa, T. Taylor, S. Oh, W. Xu, M. Gruteser, W. Trappe, and I. Seskar, “Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study,” in *USENIX Security*, 2010.

[66] C. Feng, T. Li, and D. Chana, “Multi-level anomaly detection in industrial control systems via package signatures and LSTM networks,” in *DSN*, 2017.

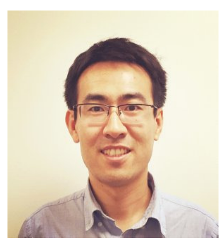
[67] S. McLaughlin, “CPS: Stateful policy enforcement for control system device usage,” in *ACSAC*, 2013, pp. 109–118.

[68] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, “S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems,” in *HiCoNS*, 2013.

[69] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, “Securecore: A multicore-based intrusion detection architecture for real-time embedded systems,” in *RTAS*, 2013.

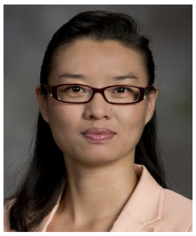
[70] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, “Drammer: Deterministic rowhammer attacks on mobile platforms,” in *CCS ’16*, 2016, pp. 1675–1689.

[71] X. Shu, D. D. Yao, N. Ramakrishnan, and T. Jaeger, “Long-span program behavior modeling and attack detection,” *ACM Transactions on Privacy and Security*, vol. 20, no. 4, pp. 1–28, Sep. 2017.



Long Cheng is currently pursuing his second Ph.D. in the Department of Computer Science at Virginia Tech. His research interests include system and network security, cyber forensics, cyber-physical systems (CPS), mobile computing, and wireless networks. He received his first Ph.D. degree from Beijing University of Posts and Telecommunications in 2012. Dr. Cheng received the Best Paper Award from *IEEE Wireless Communications and Networking Conference (WCNC)* in 2013 and the prestigious Erasmus Mundus Scholar Award from the European Union in 2014. Dr. Cheng’s research activities span across the fields of cyber security and networking. He has published over 60 papers in peer-reviewed journals and conferences, including *IEEE Transactions on Information Forensics and Security (TIFS)*, *IEEE/ACM Transactions on Networking (ToN)*, *Transactions on Parallel and Distributed Systems (TPDS)*, *IEEE Transactions on Mobile Computing (TMC)*, *Annual Computer Security Applications Conference (ACSAC)*, *Privacy Enhancing Technologies Symposium (PETS)*, *IEEE Conference on Computer Communications (INFOCOM)*, *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, and *IEEE International Conference on Network Protocols (ICNP)*. He was invited to write a review article on enterprise data breach in Wiley’s *WIREs Data Mining and Knowledge Discovery*. Dr. Cheng has extensive experiences collaborating with researchers in the industry and academia across multiple countries. He holds a patent for his sensor network routing method.

Ke Tian is a PhD student in Department of Computer Science at Virginia Tech. He received his bachelor degree majoring information security from University of Science and Technology of China. He received the National Scholarship of China in 2012. His research interests is in cybersecurity, Android security and malware detection.



Danfeng (Daphne) Yao is an associate professor of computer science at Virginia Tech. In the past decade, she has been working on designing and developing data-driven anomaly detection techniques for securing networked systems against stealthy exploits and attacks. Her expertise also includes mobile security. Dr. Yao received her Ph.D. in Computer Science from Brown University. Dr. Yao is an Elizabeth and James E. Turner Jr. '56 Faculty Fellow and L-3 Faculty Fellow. She received the NSF CAREER Award in 2010 for her work on human-behavior driven malware detection, and the ARO Young Investigator Award for her semantic reasoning for mission-oriented security work in 2014. She has several Best Paper Awards (e.g., ICNP '12, CollaborateCom '09, and ICICS '06) and Best Poster Awards (e.g., ACM CODASPY '15). She was given the Award for Technological Innovation from Brown University in 2006. She held multiple U.S. patents for her anomaly detection technologies. Dr. Yao is an associate editor of IEEE Transactions on Dependable and Secure Computing (TDSC). She serves as PC members in numerous computer security conferences, including ACM CCS and IEEE S&P. She has over 85 peer-reviewed publications in major security and privacy conferences and journals.

Lui Sha received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 1985. He is currently a Donald B. Gillies Chair Professor of Computer Science with the University of Illinois at Urbana Champaign, Champaign, IL, USA. His work on real-time computing is supported by most of the open standards in real-time computing and has been cited as a key element in the success of many national high-technology projects including GPS upgrade, the Mars Pathfinder, and the International Space Station.

Raheem Beyah is the Motorola Foundation Professor and Associate Chair for Strategic Initiatives and Innovation in the School of Electrical and Computer Engineering at Georgia Tech, where he leads the Communications Assurance and Performance Group (CAP). His research interests include network security, wireless networks, network traffic characterization and performance, and security visualization. He received the National Science Foundation CAREER award in 2009 and was selected for DARPA's Computer Science Study Panel in 2010. He is a member of AAAS, ASEE, a lifetime member of NSBE, a senior member of IEEE, and an ACM Distinguished Scientist.