POSTER: Detection of CPS Program Anomalies by Enforcing Cyber-Physical Execution Semantics*

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

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

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

ABSTRACT

In this work, we present a new program behavior model, *i.e.*, the event-aware finite-state automaton (*e*FSA), which takes advantage of the event-driven nature of control programs in cyber-physical systems (CPS) and incorporates event checking in anomaly detection. *e*FSA provides *new detection capabilities to detect data-oriented attacks* in CPS control programs, including *attacks on control intensity* (*i.e.*, hijacked for/while-loops) and *attacks on control branch* (*i.e.*, conditional branches). We implement a prototype of our approach on Raspberry Pi and evaluate *e*FSA's performance by conducting CPS case studies. Results show that it is able to effectively detect different CPS attacks in our experiments.

KEYWORDS

Anomaly detection; Cyber-physical systems; Data-oriented attacks

1 INTRODUCTION

Control programs are critical to the proper operations of cyber-physical systems (CPS), as anomalous program behaviors can have serious consequence, or even cause devastating damages to physical systems [1]. Recent studies [1, 3] 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 and data-oriented attacks. The former exploits memory corruption vulnerabilities to divert a program's control flows. The latter manipulates data variables without violating the program's control flow integrity (CFI). Because existing CFI-based solutions are rendered defenseless under data-oriented attacks, such threats are particularly alarming in CPS.

We focus on two types of runtime data-oriented attacks against control programs. *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 or preventing a blast furnace from being shut down properly). *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 [1]). These data-oriented attacks result in inconsistencies between the physical context and program execution, where executed control-flow paths do not correspond to the observations in the physical environment. Unfortunately, there exist very few defences [1, 6] and they are ineffective to prevent both attack types due to the lack of runtime execution semantics checking.

In many instances, CPS can be modeled as event-driven control systems. We refer to events as occurrences of interest that come through the cyber-physical observation process or emitted by other entities, and trigger the execution of corresponding control actions. We present an 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 (*i.e.*, the physical context that triggers corresponding CPS program behaviors), *e*FSA detects subtle data-oriented exploits when a specific physical event is missing (*i.e.*, not observed) along with the corresponding event dependent state transition.

We implement a proof-of-concept prototype on Raspberry Pi platform. Our prototype features: i) A gray-box FSA model that examines the return addresses on the stack when system calls are made. 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, with both local and distributed event verifiers to assess the physical context. We evaluate *e*FSA's performance by conducting CPS case studies. Our results show that *e*FSA is able to successfully detect different data-oriented attacks. The runtime anomaly detector takes ~0.0001s to check each state transition in *e*FSA model, ~0.063s for the local event verification, and ~0.211s for the distributed event verification.

2 THE PROPOSED APPROACH

2.1 Attack Model

We assume that the adversary 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. We mainly focus on runtime software exploits, and thus sensor data spoofing attacks in the physical domain are out of the scope of this work. 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. We also assume the runtime

*This work has been supported in part by Security and Software Engineering Research Center (S2ERC), a NSF sponsored multi-university Industry/University Cooperative Research Center (I/UCRC). The full version of this work can be found in [2].

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10.
https://doi.org/10.1145/3133956.3138839

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 ARM's TrustZone [1].

2.2 Design Overview

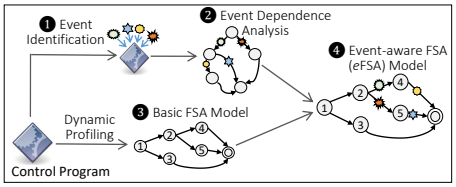


Fig. 1: *e*FSA model construction in the training phase. This workflow can be generalized to non-FSA anomaly detection frameworks (*i.e.*, augmenting an existing program behavior model with contextual integrity).

Fig. 1 shows the workflow of constructing the *e*FSA program behavior model in our design. There are four main steps in the training phase. We first identify CPS events involved in the control program (❶). After that, we perform the event dependence analysis to generate an event-annotated control flow graph (❷), which identifies event triggered instructions/statements of the program. Then, we construct the basic finite-state automaton (FSA) model based on dynamic profiling (❸). Given an event-annotated CFG, we are able to identify the event-driven system call sequences. By augmenting the event-driven information on top of the basic FSA, we generate our event-aware FSA (*i.e.*, *e*FSA) for CPS control program behavior modeling (❹).

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

2.3 Construction of *e*FSA Model

Event Identification and Dependence Analysis: In order to discover the triggering relationship between external events and internal program control flows, we first identify what events are involved in a CPS program. Without loss of generality, we define two types of events in control programs: i) binary events and ii) non-binary events. Binary events return either `True` or `False`, which are defined in terms of pre-specified status changes of physical environments and provide notifications to the control program. 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. We present an LLVM-based method for reasoning triggering relationship between external events and internal program control flows. Our key idea is to search for an LLVM branch instruction that is data-dependent on any sensor-reading API, and at least an actuation API is control-dependent on this branch instruction. The search is performed through backward data dependence analysis and forward control dependence analysis.

Event Awareness Enhancement: Our *e*FSA model extends the FSA [5] model with external context constraints, where event dependent state transitions in FSA are labeled with event constraints. The FSA construction 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, and each system call corresponds to a state transition. Then, we apply the event dependence analysis results to augment the event-driven information over the underlying FSA, and finally construct the *e*FSA model. Fig. 2(a) shows a pictorial example program. The learnt *e*FSA model is shown in Fig. 2(b), where an event dependent transition is labeled by "$[\frac{System\ Call}{PC}]$|Events". In this example, there are two binary events and one non-binary event. We identify binary-event dependent state transitions $[\frac{S_1}{3}\frac{S_2}{6}]|E_1$, $[\frac{S_1}{3}\frac{S_4}{9}]|E_2$, and a non-binary-event dependent control intensity loop $[\frac{S_2}{6}\frac{S_3}{7}]$. It also contains an implicit event dependent transition $[\frac{S_1}{3}\frac{S_5}{10}]|(\overline{E_1} \wedge \overline{E_2})$. *e*FSA expresses causal dependencies between physical events and program control flows. By checking execution semantics at runtime, *e*FSA improves the robustness against data-oriented attacks by increasing the difficulties that an attack could bypass the anomaly detection.

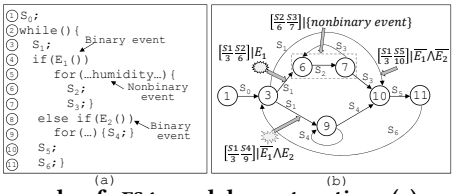


Fig. 2: Example of *e*FSA model construction: (a) an example program; (b) the corresponding *e*FSA model.

3 CPS CASE STUDY

We conduct a CPS case study, and evaluate *e*FSA's detection capability against runtime data-oriented attacks.

SyringePump[1]**.** 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 [1]. 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 dependent on the humidity value subtracted by the threshold value.

3.1 Detecting Attacks on Control Branch

In this experiment, we evaluate *e*FSA's security guarantees against control branch attacks. 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. 3 illustrates an example

[1] https://github.com/control-flow-attestation/c-flat

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. 3, *e*FSA raises an alarm when it finds a mismatch between the execution semantics and program behavior.

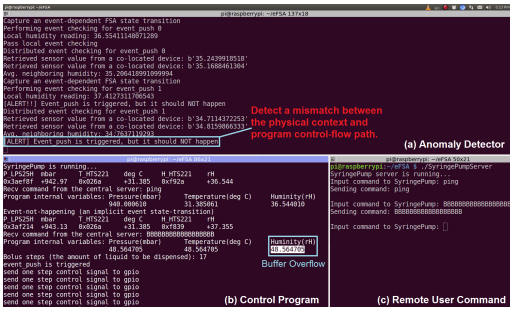
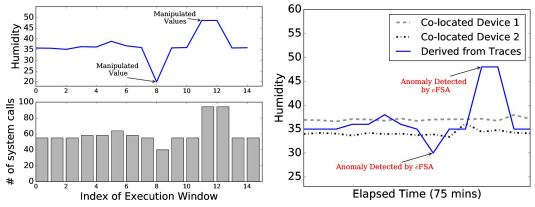


Fig. 3: An instance of detecting attacks on control branch

3.2 Detecting Attacks on Control Intensity

In this experiment, we set the threshold to 30rH, and demonstrate that *e*FSA is able to detect control intensity attacks with only system call traces. In SyringePump, 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 Syscall traces **(b) *e*FSA's detection**

Fig. 4: An instance of detecting attacks on control intensity

Fig. 4(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. 4(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.

3.3 Runtime Overhead

We measure the performance overhead incurred by *e*FSA's anomaly detector on Raspberry Pi. We employ the user-space `strace` software to collect system calls in our prototype. The system call tracing overhead has no difference between FSA and *e*FSA, incurring 1.5x∼2x overhead in our experiments. For performance consideration, alternative tracing techniques may be adopted in replacing the `strace` to improve the tracing performance. For example, authors in [4] show that the overhead of hardware-assisted system call tracing on ARM platform is sufficiently small to ignore.

The average delay out of more than 1000 runs for each state transition (*i.e.*, each intercepted system call) checking 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.

4 CONCLUSION

In this work, we present a CPS-specific program behavior model *e*FSA, which advances the state-of-the-art program behavior modelling by augmenting an existing program behavior model with physical context awareness. *e*FSA detects subtle data-oriented exploits in CPS if a specific physical event is missing along with the corresponding event dependent state transition. We implement a proof-of-concept prototype to demonstrate the feasibility of our approach. Real-world CPS case study demonstrates *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 our design paradigm to support actuation integrity for fine-grained anomaly detection at the instruction level.

REFERENCES

[1] 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.
[2] L. Cheng, K. Tain, and D. D. Yao. Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *ACSAC*, 2017.
[3] 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.
[4] Z. Ning and F. Zhang. Ninja: Towards transparent tracing and debugging on arm. In *USENIX-Security*, 2017.
[5] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Security and Privacy*, 2001.
[6] 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 *IoTDI*, 2017.