Program Anomaly Detection Against Data-Oriented Attacks

Long Cheng

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science & Application

Danfeng (Daphne) Yao, Chair
Raheem Beyah
Patrick Schaumont
Naren Ramakrishnan
Gang Wang

July 26, 2018
Blacksburg, Virginia

Keywords: Program Anomaly Detection, Data-Oriented Attacks, Cyber-Physical Systems
Copyright 2018, Long Cheng
Program Anomaly Detection Against Data-Oriented Attacks

Long Cheng

(ABSTRACT)

Memory-corruption vulnerability is one of the most common attack vectors used to compromise computer systems. Such vulnerabilities could lead to serious security problems and would remain an unsolved problem for a long time. Existing memory corruption attacks can be broadly classified into two categories: i) control-flow attacks and ii) data-oriented attacks. Though data-oriented attacks are known for a long time, the threats have not been adequately addressed. As launching a control-flow attack becomes increasingly difficult due to many deployed defenses against control-flow hijacking, data-oriented attacks are considered an appealing attack technique for system compromise, including the emerging embedded control systems.

To counter data-oriented attacks, mitigation techniques such as memory safety enforcement and data randomization can be applied in different stages over the course of an attack. However, attacks are still possible because currently deployed defenses can be bypassed. This dissertation explores the possibility of defeating data-oriented attacks through external monitoring using program anomaly detection techniques. I start with a systematization of current knowledge about exploitation techniques of data-oriented attacks and the applicable defense mechanisms. Then, I address three research problems in program anomaly detection against data-oriented attacks.

First, I address the problem of securing control programs in Cyber-Physical Systems (CPS) against data-oriented attacks. I describe a new security methodology that leverages the event-driven nature in characterizing CPS control program behaviors. By enforcing runtime cyber-physical execution semantics, our method detects data-oriented exploits when physical events are inconsistent with the runtime program behaviors.

Second, I present a statistical program behavior modeling framework for frequency anomaly detection, where frequency anomaly is the direct consequence of many non-control-data attacks. Specifically, I describe two statistical program behavior models, $sFSA$ and $sCFT$, at different granularities. Our method combines the local and long-range models to improve the robustness against data-oriented attacks and significantly increase the difficulties that an attack bypasses the anomaly detection system.

Third, I focus on defending against data-oriented programming (DOP) attacks using Intel Processor Trace (PT). DOP is a recently proposed advanced technique to construct expressive non-control data exploits. I first demystify the DOP exploitation technique and show its complexity and rich expressiveness. Then, I design and implement the DeDOP anomaly detection system, and demonstrate its detection capability against the real-world ProFTPd DOP attack.
Program Anomaly Detection Against Data-Oriented Attacks

Long Cheng

(GENERAL AUDIENCE ABSTRACT)

Memory-corruption vulnerability is one of the most common attack vectors used to compromise computer systems. Such vulnerabilities could lead to serious security problems and would remain an unsolved problem for a long time. This is because low-level memory-unsafe languages (e.g., C/C++) are still in use today for interoperability and speed performance purposes, and remain common sources of security vulnerabilities. Existing memory corruption attacks can be broadly classified into two categories: i) control-flow attacks that corrupt control data (e.g., return address or code pointer) in the memory space to divert the program’s control-flow; and ii) data-oriented attacks that target at manipulating non-control data to alter a program’s benign behaviors without violating its control-flow integrity.

Though data-oriented attacks are known for a long time, the threats have not been adequately addressed due to the fact that most previous defense mechanisms focus on preventing control-flow exploits. As launching a control-flow attack becomes increasingly difficult due to many deployed defenses against control-flow hijacking, data-oriented attacks are considered an appealing attack technique for system compromise, including the emerging embedded control systems. To counter data-oriented attacks, mitigation techniques such as memory safety enforcement and data randomization can be applied in different stages over the course of an attack. However, attacks are still possible because currently deployed defenses can be bypassed.

This dissertation explores the possibility of defeating data-oriented attacks through external monitoring using program anomaly detection techniques. I start with a systematization of current knowledge about exploitation techniques of data-oriented attacks and the applicable defense mechanisms. Then, I address three research problems in program anomaly detection against data-oriented attacks. First, I address the problem of securing control programs in Cyber-Physical Systems (CPS) against data-oriented attacks. The key idea is to detect subtle data-oriented exploits in CPS when physical events are inconsistent with the runtime program behaviors. Second, I present a statistical program behavior modeling framework for frequency anomaly detection, where frequency anomaly is often consequences of many non-control-data attacks. Our method combines the local and long-range models to improve the robustness against data-oriented attacks and significantly increase the difficulties that an attack bypasses the anomaly detection system. Third, I focus on defending against data-oriented programming (DOP) attacks using Intel Processor Trace (PT). I design and implement the DeDOP anomaly detection system, and demonstrate its detection capability against the real-world DOP attack.
Acknowledgments

I would like to express my deepest gratitude to my advisor, Dr. Danfeng (Daphne) Yao, for her guidance, support and trust throughout my graduate study at Virginia Tech. I very much appreciate that Dr. Yao introduced me to the field of cyber security research, shared me with her profound knowledge, and navigated me to pursue an academic career. Over the past three years, I have learned so much from Dr. Yao and the Yao group. Without her encouragement and inspiration, this dissertation would not have been possible. I feel lucky to have Dr. Yao as my advisor and I sincerely hope that we will continue our academic collaborations in the future.

I would also like to express my sincere gratitude to the committee members, Dr. Raheem Beyah, Dr. Patrick Schaumont, Dr. Naren Ramakrishnan, Dr. Gang Wang, for their insightful comments and valuable suggestions on my research. I am also very grateful to many academic collaborators, Dr. Trent Jaeger, Dr. N. Asokan, Dr. Lui Sha, Nyman Thomas, Liljestrand Hans, and Ben Heidorn, for their collaborative efforts in our joint projects.

It has been a pleasure to get to know and work with so many smart friends in CRC KWII: Dr. Hao Zhang, Dr. Xiaokui Shu, Dr. Fang Liu, Dr. Ke Tian, Sazzadur Rahaman, Alexander Kedrowitsch, Xiaodong Yu, Ya Xiao, Md Salman Ahmed, Dr. Yue Cheng, Dr. Haipeng Cai, Dr. Zheng Song, Tong Zhang, Qingrui Liu, Bo Li, Xinwei Fu, Yin Liu, Run Yu, Kaidi Wang, Spencer Lee, Peeratham Techapalokul, Stefan Nagy, etc. Their companionship has made my graduate school life in Blacksburg enjoyable and memorable.

I am greatly indebted to my parents and my wife Xiaoyan Han for their never-ending support and love. My daughters Emily Cheng and Hannah Cheng have brought so much joy to our lives. My family gives me the strongest motivation to keep working hard.

My thesis work has been supported in part by the Office of Naval Research under Grant ONR-N00014-17-1-2498, DARPA/ONR under Grant N66001-17-C-4052, and National Science Foundation under Grant OAC-1541105.
## Contents

1 Introduction ............................................. 1
   1.1 Motivation ........................................... 1
   1.2 Event-Aware Program Anomaly Detection in Cyber-Physical Systems ............................................. 3
   1.3 Statistical Program Behavior Modeling for Frequency Anomaly Detection ............................................. 4
   1.4 Defending Against Data-Oriented Programming (DOP) Attacks ............................................. 5
   1.5 Dissertation Roadmap ............................................. 6

2 Literature Review ............................................. 7
   2.1 Data-Oriented Attacks ............................................. 7
      2.1.1 Classification of Data-Oriented Attacks ............................................. 7
      2.1.2 Three-Stage Model of Data-Oriented Attacks ............................................. 9
   2.2 Defense Mechanisms Against Data-Oriented Attacks ............................................. 12
      2.2.1 S1 Defense ............................................. 12
      2.2.2 S2 Defense ............................................. 16
      2.2.3 Defenses Specific to Data-Oriented Attacks ............................................. 17
   2.3 Program Anomaly Detection ............................................. 19
      2.3.1 Program Behavior Modeling ............................................. 20
3 Event-Aware Program Anomaly Detection in Cyber-Physical Systems

3.1 Instruction ......................................................... 30
3.2 Model and Design Overview ................................. 32
  3.2.1 CPS Background ........................................... 33
  3.2.2 Attack Model and Assumptions ........................ 34
  3.2.3 New Detection Capabilities ............................. 35
  3.2.4 Definition of Events ....................................... 36
  3.2.5 Orpheus Framework Overview .......................... 36
3.3 Reasoning About Cyber-Physical Execution Semantics ................. 37
  3.3.1 Event Identification ....................................... 37
  3.3.2 Event Dependence Analysis .............................. 39
3.4 eFSA: an Instantiation of Orpheus .......................... 41
  3.4.1 Formal Description of eFSA .............................. 41
  3.4.2 From Event-Annotated CFG to eFSA ................... 42
  3.4.3 Security Policies in eFSA ............................... 43
  3.4.4 Control Intensity Analysis .............................. 44
3.5 EFSA-based Detection ........................................... 45
  3.5.1 Runtime Monitoring and Detection ..................... 45
  3.5.2 Event Verification Strategies .......................... 46
3.6 Implementation .................................................. 46
3.7 Experimental Validation ....................................... 48
  3.7.1 CPS Case Studies ......................................... 48
  3.7.2 Training and Runtime Performance ..................... 49
List of Figures

2.1 Stages in data-oriented attacks and mitigations in different stages . . . . . . . . . . 10
2.2 Training phase and monitoring phase in program anomaly detection . . . . . . . . 20

3.1 An abstract view of the event-driven CPS architecture. . . . . . . . . . . . . . . . 33
3.2 Two examples of data-oriented software exploits in a real-world CPS application . 35
3.3 Workflow of Orpheus event-aware anomaly detection framework, which augments an existing program behavior model with cyber-physical contextual integrity. . . . 37
3.4 An example of identifying non-binary events . . . . . . . . . . . . . . . . . . . . . 39
3.5 Event dependence analysis for nested events . . . . . . . . . . . . . . . . . . . . . . 41
3.6 System-call based finite-state automaton (FSA) model: (a) an example program; (b) the corresponding FSA model. . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.7 An example of the eFSA model . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.8 eFSA-based anomaly detection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.9 An example of using strace tool with stack unwinding support, where call stacks are printed out with the system-call. . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
3.10 An instance of Solard experiment . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
3.11 An instance of SyringePump experiment . . . . . . . . . . . . . . . . . . . . . . . 52
3.12 An instance of SyringePump experiment with a sampling rate of 5 minutes . . . . . 53

4.1 Statistical program behavior modeling framework . . . . . . . . . . . . . . . . . . . 60
4.2 (a) FSA augmented with state transition frequencies; (b) Calculation of n-step state transition probabilities . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.3 Behavior instances are transformed into cdf curves and then clustering is performed.

4.4 An example that simplifies the attack on the authentication flag in SSH server [1].

4.5 An example of program execution traces (i.e., user space PT events) collected by PT corresponding to the code in Fig. 4.4.

4.6 Examples of TNT packets and the corresponding assembly code.

4.7 sFSA transforms program behaviors to cdf curves. The anomalous program behaviors exhibit distinctive patterns from the normal ones.

4.8 Tradeoff between detection rate and false positives.

4.9 Impact of step length.

4.10 Training overhead.

4.11 X-clustering for 2-step control-flow-transfers with PCA reduction to 3-dimension.

4.12 Tracing overhead.

5.1 ProFTPd DOP attack flow.

5.2 Three types of anomalous control-flow behaviors caused in DOP attacks.

5.3 Workflow of DEDOP anomaly detection system.

5.4 Examples of direct and indirect branch correlations.

5.5 Example of inter-procedure branch correlation.

5.6 Example of an incompatible branch behavior.

5.7 X-clustering for 2-step control-transfers with PCA reduction to 3-dimension in sreplace. The DOP attack exhibits a unique pattern of control-flow transfers in comparison to baseline executions.

5.8 2-gram distribution of FTP commands during the ProFTPd DOP attack.

5.9 X-clustering for 2-grams of FTP commands with PCA reduction to 3-dimension using LBNL-FTP-PKT dataset [2]. The DOP attack involves an abnormally high number of client-server interactions.

5.10 Tracing overhead of sreplace.
# List of Tables

2.1 Comparison of defensive mechanisms against data-oriented attacks . . . . . . . . 17

2.2 Tracing overhead of different software-based system-call tracing tools . . . . . . . 22

2.3 Comparison of representative CPS anomaly detection approaches . . . . . . . . . 26

3.1 Average delay overhead in training phase . . . . . . . . . . . . . . . . . . . . . . 50

3.2 Runtime overhead in the monitoring phase . . . . . . . . . . . . . . . . . . . . . . 50

4.1 Number of evasive cases (i.e., attack freedom) of different program behavior models 59

4.2 Summary of sFSA and sCFT models . . . . . . . . . . . . . . . . . . . . . . . . . 70

5.1 Branch correlation analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
Chapter 1

Introduction

1.1 Motivation

Exploiting software’s memory-corruption vulnerabilities is one of the most common attack methods to compromise computer systems [3]. Such vulnerabilities could lead to serious security problems and would remain an unsolved problem for a long time. This is because: 1) memory-unsafe languages such as C and C++ are still widely used today and remain common sources of security vulnerabilities; and 2) despite considerable research in past years, existing mechanisms suffer one or more issues including large performance overhead, inaccurate/incomplete coverage, or imperfect compatibility [4]. Low-overhead approaches usually offer inadequate protection/coverage, while comprehensive solutions either incur a large performance overhead or provide a limited backward compatibility [5].

Memory-corruption vulnerabilities can be exploited in different ways, potentially resulting in arbitrary code execution and data manipulation. Existing software exploits can be broadly classified into two categories: i) control-flow attacks and ii) data-oriented attacks\(^1\).

**Control-flow attacks** corrupt control data (e.g., return address or code pointer) in a program’s memory space to divert the program’s control-flow, including malicious code injection [6], code reuse attacks [7], return oriented programming (ROP) attacks [8]. To counter these attacks, many defense mechanisms have been proposed, such as stack canaries [9], Address Space Layout Randomization (ASLR) [10], Data Execution Prevention (DEP) [11], Control-Flow Integrity (CFI) [12], Intel’s CET [13] and MPX [14], and Return Flow Guard (RFG) [15]. In particular, CFI-based solutions

---

\(^1\)In this thesis, we focus our investigation on data-oriented attacks that are caused by memory-corruption vulnerabilities. Data-only attacks can also be caused by hardware transient faults, logic errors in code (e.g., the Heartbleed bug without buffer overflow), or data spoofing/injection attacks in the physical domain for sensor-driven programs.
have received considerable attention in the last few years. They enforce security policies on all indirect control transfer instructions *(e.g., ret/jmp)*, and thus ensure that the program execution always follows a valid path in the Control-Flow Graph (CFG).

**Data-oriented attacks** can change the program’s benign behavior by manipulating the program’s non-control data without violating its control-flow integrity. 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. The attack objections include: 1) information disclosure *(e.g., leaking passwords, private keys or sensitive randomized values)*, 2) privilege escalation *(e.g., by manipulating user identity data)*, and 3) performance degradation *(e.g., resource wastage attack)* [16]. Data-oriented attacks can be as simple as flipping a bit of a data variable. But they can be equally powerful and effective as control-flow attacks [17], *e.g.*, arbitrary code-execution attacks are possible if an attacker could corrupt parameters of system-calls *(e.g., execve())* [3]. Though data-oriented attacks are known for a long time, the threats have not been adequately addressed due to the fact that most previous defense mechanisms focus on preventing control-flow exploits.

As launching a control-flow attack becomes increasingly difficult due to many deployed defenses against control-flow hijacking, data-oriented attacks are likely to become an appealing attack technique for system compromise [18, 19, 20, 21, 22, 23]. Rogowski *et al.* [24] showed the practicality of data-only exploits against Internet Explorer and Chrome. Morton *et al.* [23] demonstrated the feasibility of data-oriented attacks against Nginx and Apache web servers. For example, by manipulating only a few bytes in memory, it allows an attacker to re-configure a running web server on the fly to degrade or disable services. Davi *et al.* [25] showed that a data-only attack on page tables can undermine the kernel CFI protection. By manipulating the memory permissions in kernel page entries, the attack makes kernel code pages writable and subsequently (re)enables malicious code injection to kernel space. Jia *et al.* [26] exploited data-oriented attacks to bypass the same-origin policy (SOP) enforcement in Chrome browser.

To counter data-oriented attacks, mitigation techniques such as memory safety enforcement [27], data randomization [10], Data-Flow Integrity (DFI) [28] can be applied in different stages over the course of an attack. However, real world software exploits are still possible because currently deployed defenses can be bypassed [5]. Program anomaly detection complements existing mitigation techniques, and can serve as the last line of defense against data-oriented attacks.

This thesis presents attempts to defeat data-oriented attacks through program anomaly detection techniques. We start with a systematization of current knowledge about exploitation techniques of data-oriented attacks and the applicable defense mechanisms. Then, we address three problems in program anomaly detection against data-oriented attacks: 1) enforcing cyber-physical execution semantics to defend against data-oriented attacks in Cyber-Physical Systems (CPS); 2) statistical pro-
gram behavior modeling for frequency anomaly detection; and 3) defending against data-oriented programming (DOP) attacks using Intel Processor Trace (PT).

In the rest of this chapter, I briefly introduce the research problems and my technical contributions to the areas of program anomaly detection. I hope that this thesis will stimulate a broader discussion about advanced defenses against data-oriented attacks.

1.2 Event-Aware Program Anomaly Detection in Cyber-Physical Systems

Cyber-physical systems (CPS) consist of a tightly coupled integration of computational elements and physical components. It has received considerable attention due to the wide range of applications, such as electric power grid, oil and gas distribution, industrial automation, medical devices, automobile systems, air traffic control, and smart homes [29]. Control programs running on CPS devices monitor physical environments by taking sensory data as input and send control signals that affect physical systems with feedback loops [30, 31]. They are critical to the proper operations of CPS, as anomalous program behaviors can have serious consequences, or even cause devastating damages to physical systems [32]. For example, the Stuxnet [33] 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 [34], there have been continuously an 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 [35, 36, 37, 32, 38, 22] have shown that control programs in embedded systems suffer from a variety of runtime software exploits. Program anomaly detection has been considered as an effective defense mechanism against software exploits. It implements an application-level reference monitor by modeling normal program behavior. Such monitoring and detection approach possesses two favourable properties: 1) it requires no modification of the binary code and thus supports legacy programs/applications; and 2) it detects a wide range of attacks/errors, e.g., not only memory corruption attacks, but also hardware/logic errors, etc. Existing program anomaly detection models [39] mainly focus on control-flow integrity checking, and thus cannot detect runtime data-oriented attacks.

In this thesis, I propose Orpheus, a new security methodology for defending against data-oriented attacks by enforcing cyber-physical execution semantics. I first present a general method for reasoning about 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 de-
pendence analysis. As an instantiation of Orpheus, I present a new program behavior model, i.e., the
event-aware finite-state automaton (eFSA). eFSA takes advantage of the event-driven nature of CPS
control programs and incorporates event checking in anomaly detection. It detects data-oriented
exploits if a specific physical event is missing along with the corresponding event dependent state
transition. I implement a proof-of-concept prototype on Raspberry Pi platforms, and evaluate the
prototype’s performance by conducting case studies under data-oriented attacks. Results show that
eFSA can successfully detect different runtime attacks. The prototype on Raspberry Pi incurs a
low overhead, taking 0.0001s for each state transition integrity checking, and 0.063s∼0.211s for
the cyber-physical contextual consistency checking.

1.3 Statistical Program Behavior Modeling for Frequency
Anomaly Detection

Frequency anomaly refers to anomalous program behavior with aberrant occurrence frequencies
of system events (e.g., system-calls, library-calls, function-calls, or control-transfers) [40]. Such
anomalous patterns are the direct consequences of many data-oriented attacks. For example, cor-
ruping non-control-data variables which directly or indirectly affect the amount of loop iterations
can lead to abnormal patterns of the loop usage. Frequency anomaly does not violate CFI and may
not introduce any unknown short call sequences (i.e., n-grams). Therefore, local detection models
that capture local execution context features such as n-gram or state-based models [41] can not
efficiently detect frequency anomalies. On the other hand, existing long-range detection models
(e.g., distribution-based [42] or co-occurrence-based [40] models) neglect the temporal relations
among neighbouring system events. Since attackers often attempt to evade or bypass the deployed
detection, such long-range models provide insufficient security against mimicry attacks.

In this thesis, I present a statistical program behavior modeling framework, which combines the
local and long-range detection models to improve the robustness against mimicry attacks. Through
numerical analysis, I demonstrate that the statistical program behavior modeling significantly in-
creases the difficulties that an attack bypasses an anomaly detection system. The key idea is to
model normal behaviors of a program by frequency distributions of short program execution se-
quences on top of underlying local models such as n-gram or FSA [41]. It captures aberrant occur-
rence frequencies hidden in long-distance system events, but also preserves local temporal relations
among adjacent system events. Then, cluster analysis is used to learn distinct normal program be-
haviors from a set of distributions. I describe two statistical program behavior models: 1) sFSA at
the system-call level; and 2) sCFT at the control-flow-transfer level (where the control-flow trac-
To improve the sFSA’s scalability, I propose a novel method to project high dimensional n-gram system-call distributions onto a line and perform cluster analysis in the two-dimensional space. sCFT models the frequency distributions of n-step control-flow transfers of a program execution. It further raises the security bar for more accurate and precise program behavior modeling. I implement prototypes of the proposed approaches and conduct an extensive experimental evaluation to demonstrate their effectiveness against real-world data-oriented attacks and synthetic data-oriented anomalies.

1.4 Defending Against Data-Oriented Programming (DOP) Attacks

Data-oriented programming (DOP) [21] is a recently proposed advanced technique to construct expressive non-control data exploits. Similar to the Return-Oriented Programming (ROP) [8], it allows an attacker to perform expressive (Turing-complete) computations in program memory by chaining the execution of short sequences of instructions (i.e., termed as data-oriented gadgets). Different from basic non-control-data attacks in which an attacker directly manipulates the target data in memory space to accomplish the malicious goal [1], data-oriented gadgets are re-engineered for data manipulation by misinterpreting existing short code sequences.

Program tracing is an important supporting technique to enable program anomaly detection, which involves recording information about a program’s execution. Tracing has been a serious performance bottleneck for program anomaly detection, hindering its deployment in practice. Intel Processor Trace (PT) is a commercially available low-overhead hardware feature on Intel CPUs, which has received considerable attention because it has the potential to bring anomaly detection to the practical deployment. PT is able to capture complete control-flow information of program execution, and thus provides a finer tracing granularity than system-call or function-call tracing. More recently, several research efforts explore the effectiveness of PT for CFI enforcement [43, 44, 45, 46]. To the best of my knowledge, the applicability of using PT for detecting data-oriented attacks has not been addressed in the literature.

In this thesis, I focus on defending against DOP attacks using PT tracing. I analyze the limits of protection offered by branch-level (i.e., PT-based) program anomaly detection against data-oriented attacks. I dissect a real-world DOP attack into steps to demystify the DOP exploitation technique and show its complexity and rich expressiveness. A DOP exploit typically involves multiple-step data manipulations. I observe that DOP attacks may manifest anomalous control-flow behaviors in three aspects: 1) incompatible branch behavior; 2) unusual execution frequency of short control-flow paths; and 3) interaction frequency anomaly. I design and implement the DE-DOP anomaly de-
tection system. DEDOP enforces the branch correlation integrity and improves anomaly detection sensitivity by capturing statistical characteristics of both short control-transfers and client-server interactions (for server applications). In particular, I develop a general branch correlation analysis tool based on LLVM [47]. This analysis tool automatically identifies coarse-grained correlated branches that have either direct or indirect joint data-dependency. Through experimental validation, I show that though DOP can be used to construct expressive (Turing-complete) non-control data attacks, they are less evasive from detection than the basic non-control-data attacks. Results demonstrate that DEDOP can successfully detect anomalies in different dimensions against the real-world ProFTPd DOP attack, and it can also be applied to detect general data-oriented attacks.

1.5 Dissertation Roadmap

The structure of the rest of this thesis is as follows. In Chapter 2, I provide a systematization of current knowledge about exploitation techniques of data-oriented attacks and the applicable defense mechanisms. Chapter 3 presents my work on event-aware program anomaly detection in CPS. The statistical program behavior modeling for frequency anomaly detection is introduced in Chapter 4. Chapter 5 presents the DEDOP detection method to detect anomalous program behaviors caused by DOP attacks. Chapter 6 concludes the thesis and discusses future research directions.
Chapter 2

Literature Review

This chapter reviews the literature related to the thesis. In Sections 2.1 and 2.2, we systematize the current knowledge about exploitation techniques of data-oriented attacks and the applicable defense mechanisms. Section 2.3 reviews related works on program anomaly detection. Section 2.4 summarizes related works on anomaly detection in Cyber-Physical Systems (CPS).

2.1 Data-Oriented Attacks

2.1.1 Classification of Data-Oriented Attacks

We classify data-oriented attacks into two categories based on how attackers manipulate the non-control data in memory space: 1) Direct Data Manipulation (DDM) attacks; and 2) Data-Oriented Programming (DOP) attacks.

1) DDM refers to a category of attacks in which an attacker directly manipulates the target data to accomplish the malicious goal. It requires the attacker to know the precise memory address of the target non-control-data. The address to a known location utilized in the attack can be derived directly from the binary analysis (e.g., a global variable with a deterministic address) or by reusing the runtime randomized address stored in memory [19]. Several types of memory corruption vulnerabilities, e.g., format string vulnerabilities, buffer overflows, integer overflows, and double free vulnerabilities [48], can allow attackers to directly overwrite arbitrary memory locations within the address space of a vulnerable application. Chen et al. [1] first revealed that DDM attacks can corrupt a variety of security-critical data variables including user identity data, configuration data, user input data, and decision-making data, which change the program’s benign behavior or cause
the program to inadvertently leak sensitive data. Code 2.1 shows an example of the non-control data attack in a vulnerable FTP server Wu-FTPd, which was reported by Chen et al. [1]. The attacker can overwrite the global data pw->pw_uid (i.e., representing the user’s UID) with the root user’s id by utilizing a format string vulnerability (in line 2). It enables the attacker to gain the root user privilege after calling seteuid in line 5.

```c
1 pw->pw_uid = getuid(); //get normal uid
2 printf(...);
3 //format string error, corrupt pw->pw_uid
4 ...
5 seteuid(pw->pw_uid); //use the corrupted data
```


2) **DOP** is an advanced technique to construct expressive non-control data exploits [21]. Similar to Return-Oriented Programming (ROP), it allows an attacker to perform expressive (Turing-complete) computations in program memory by chaining the execution of short sequences of instructions (i.e., termed as data-oriented gadgets) that are re-engineered for malicious purposes in a sequence. Typically, such a short sequence of instructions is composed of the load, store and some arithmetic micro-operations. There are two features that distinguish DOP from DDM: 1) Data-oriented gadgets are used for data manipulation by misinterpreting existing short code sequences, not only directly modifying non-control data in memory space; and 2) These gadgets are chained together by one or more dispatchers during the attack to achieve the desired outcome, which serve as the building blocks in a DOP attack.

```c
1 struct server{int *cur_max, total, typ;} *srv;
2 int connect_limit = MAXCONN; int *size, *type;
3 char buf[MAXLEN];
4 size = &buf[8]; type = &buf[12];
5 ...
6 while(connect_limit--) {
7   readData(sockfd, buf); // stack buffer overflow
8   if(*type == NONE ) break;
9   if(*type == STREAM)
10      *size = *(srv->cur_max); //dereference gadget
11   else {
12      srv->typ = *type; //assignment gadget
13      srv->total += *size; //addition gadget
14   } ...
15 }
```

Code 2.2: Example of data-oriented gadgets in DOP. The code snippet models a vulnerable FTP server.
Code 2.2 is an example to introduce the concept of data-oriented gadgets [21]. The code snippet models an FTP server, which receives network requests and processes them based on message types. It is assumed that a buffer overflow vulnerability exists in function readData in line 7, which does not perform the bounds check of buf. As a result, the values of local variables including srv, connect_limit, size, and type can be corrupted. Three micro-operations in Code 2.2 can be used as data-oriented gadgets: 1) size=*(srv->cur_max) in line 10 as a dereference gadget; 2) srv->typ=*type in line 12 as an assignment gadget; and 3) srv->total+=*size in line 13 as an addition gadget. Since these micro-operations are within the while loop in lines 6 to 15, and the loop iteration variable connect_limit is under the control of an attacker, they can be chained sequentially and misinterpreted for malicious purposes by a remote adversary. The while loop is called as a gadget dispatcher. In [21], Hu et al. illustrated that data-oriented gadgets in Code 2.2 can be re-engineered to perform an expressive calculation on behalf of the attacker, such as a linked list update function.

Constructing DOP attacks is non-trivial in real-world programs, including identifying data-oriented gadgets and gadget dispatchers. Then, the attacker needs to craft malicious payloads to trigger the execution of selected gadgets with the expected addresses and sequence.

### 2.1.2 Three-Stage Model of Data-Oriented Attacks

We present a *three-stage* taxonomy of existing defense techniques against data-oriented attacks. Fig. 2.1 illustrates the abstract view of the three stages in data-oriented attacks. To launch such attacks, it starts with triggering a memory error of a vulnerable program (*i.e.*, Stage S1), which empowers an attacker with control to the memory space, *e.g.*, read/write capability. In Stage S2, the targeted non-control-data is modified (through either DDM or DOP). In Stage S3, the manipulated data variable is used and takes effect to change the default program behavior. Note that S3 does not necessarily happen immediately after the data manipulation. The back edges pointing from S3→S1 and S2→S1 indicate that an attacker may need to corrupt non-control-data multiple times to achieve the malicious goal. Typically, a DOP attack corrupts substantial memory locations in a program and involves multiple steps of non-control data manipulation.

We list requirements in different stages (*i.e.*, the threat model) that are essential for launching a successful data-oriented attack. The first three requirements apply for DDM exploits. While DOP attacks need to satisfy all these requirements.

* The presence of a memory corruption vulnerability (such as a buffer or heap overflow) in the target program, which allows attackers to modify the content of the application’s memory (*i.e.*, write capability). This is a reasonable assumption since low-level memory-unsafe languages (*e.g.*,
C/C++) are still in use today for interoperability and speed performance purposes, even though the memory corruption vulnerability is an inevitable security weakness in these languages.

* Knowing the exact location of target non-control data in memory. Due to the wide deployment of exploit mitigation technologies such as DEP and ASLR, it is likely attackers need to first leverage memory disclosure vulnerabilities to circumvent the address space randomization [23]. In this case, a covert exfiltration channel to achieve information leakage is needed (i.e., read capability), such as reading data from arbitrary addresses of the target program.

* Knowing exactly the transformation from an attack payload to the impact on memory space of the target program, to avoid program crash and CFI violation.

* Availability of DOP gadgets that are reachable by the memory corruption vulnerability, and triggerable by the attack payload.

* Stitchability of disjoint DOP gadgets. One or more gadget dispatchers are needed to dispatch and execute the functional DOP gadgets.

Attackers need to satisfy the above requirements to launch data-oriented exploits. It allows different defense mechanisms to prevent/detect data-oriented attacks at multiple points/stages. We first provide an overview of defenses focusing on preventing these requirements from been satisfied. We discuss defense mechanisms in details in Section 2.2.

**S1 Defense – Preventing Exploitation of Memory Errors**

*Memory safety* enforcement is the first line of defense, which aims to prevent both spatial and temporal memory errors, such as buffer overflows and use-after-free errors. Enforcing strict mem-
ory safety could fundamentally solve the memory corruption issue, which prevents all unexpected
memory reads and writes. However, it incurs a high performance overhead even with hardware
support [5].

**S2 Defense – Providing a Barrier to Access to Data or Guess Memory Layout**

The purpose of S2 defenses is to mitigate the consequences of attacks in the presence of memory
vulnerabilities. S2 defenses include *software compartmentalization* [49, 50, 51] and *address space
or data layout randomization* [10, 52] techniques. They serve as the second line of defense, which
creates a barrier for attackers to access to the target non-control data or guess the memory layout.

Software compartmentalization isolates software components into distinct protection domains in
order to limit the utility of existing memory errors (*i.e.*, when the memory error and data to be
manipulated exist in different protection domains), but also limit the abilities of a compromised
software component. Randomization techniques such as Address Space Layout Randomization
(ASLR) [10] and Data Space Randomization (DSR) [52] provide probabilistic protections against
data manipulation. ASLR randomly chooses the base addresses of stack, heap, code segment,
and shared libraries. DSR encrypts data stored in memory, rather than randomizing the location.
Though strong randomization can stop memory corruption attacks with a high probability, the pro-
tection is confined to data/addresses that are randomized/encrypted. In practice, to prevent per-
formance degradation, not all data/addresses are protected by randomization defenses [5]. On the
other hand, information leaks can also undermine against these randomization techniques. In addi-
tion, data/address encryption based solutions are not binary compatible (*i.e.*, protected binaries are
incompatible with unmodified libraries) [5].

**S3 Defense – Preventing/Detecting the Use of Corrupted Data**

Data-Flow Integrity (DFI) [28] mitigates data corruption before the manipulation takes effect. Be-
fore each read instruction, DFI ensures that a variable can only be written by a legitimate write
instruction which can be derived by reaching definition analysis (*i.e.*, for each value read instruc-
tion, it statically computes the set of write instructions that may write the value). However, DFI
usually overestimates the set of valid write instructions since the set is statically determined without
runtime information. Moreover, Software-based DFI incurs a high performance overhead [21] due
to the frequent read instruction checking. Hardware-based DFI, *e.g.*, HDFI [53], is efficient, but
limited by the number of simultaneous protection domains it can support. Carlini *et al.* [3] have
recently revealed fundamental limits on the effectiveness of CFI, and presented the Control-Flow
Bending (CFB) which allows an attacker to "bend" the control-flow of a program but adheres to
CFI’s security policies, *i.e.*, modifying indirect branch targets that are valid based on a CFI policy. Depending on the granularity of compartmentalization, and the boundaries of the security domain, software compartmentalization can also function as a defense in S3. It can be used to prevent the use of corrupted data. For example, when a corrupted pointer is referencing memory in another protection domain, it thwarts the dereference operation in S3.

**S1-S3’s Complementary Defense – Program Anomaly Detection**

Szekeres et al. [5] provided a systematical overview of memory corruption attacks and mitigations. They highlighted that though a vast number of solutions have been proposed, memory corruption attacks continue to pose a serious security threat. Real-world software exploits are still possible because currently deployed defenses can be bypassed. *Program anomaly detection* complements the mitigation techniques aforementioned, and serves as the last line of defense against data-oriented attacks. As shown in Fig. 2.1, program anomaly detection has the potential to detect anomalous program behaviors exhibited in all the three stages in data-oriented attacks. We introduce program anomaly detection in details in Section 2.3.

**2.2 Defense Mechanisms Against Data-Oriented Attacks**

We first discuss representative generic memory corruption prevention mechanisms in S1 and S2. These defense techniques can prevent general types of memory corruption attacks, which apply for both control-flow attacks and data-oriented attacks. In addition to generic memory corruption prevention mechanisms, a number of detection and prevention techniques have been proposed in the literature. We then discuss defensive mechanisms especially focusing on data-oriented attacks.

**2.2.1 S1 Defense**

S1 defenses (*i.e.*, *memory safety*) protect from security vulnerabilities dealing with memory accesses. It ensures the low-level integrity of a program’s data structures, avoiding both spatial and temporal invalid memory accesses. Memory-safe programming languages achieve this with built-in runtime bounds checks and garbage collection that make them immune to memory errors. In contrast, *memory-unsafe* languages such as C and C++ lack built-in memory safety guarantees. Programs written in memory-unsafe languages therefore commonly exhibit memory errors that may make them vulnerable to runtime exploitation. Nevertheless, C and C++ are still widely used programming languages today. Despite considerable prior research in retrofitting memory-unsafe
programs with memory safety guarantees, memory-safety problems persist due to a trade-off between effectiveness and efficiency: low-overhead approaches usually offer inadequate protection/coverage, while comprehensive solutions either incur a high performance overhead or provide a limited backward compatibility.

Enforcing all memory accesses staying within the bounds of intended objects would completely eliminate the pre-conditions for all attacks that rely on gaining access to a prohibited area of memory. Memory-unsafe languages allow direct access to memory using pointers, which is a common cause of memory corruption. The majority of existing memory safety proposals can be generally classified into two categories: pointer safety (i.e., pointer-based approaches focusing on pointer dereference operations, including pointer-based bounds checking and pointer integrity/authenticity) and object safety (i.e., object-based approaches focusing on pointer arithmetic operations) [5, 4].

**Pointer-Based Bounds Checking**

Pointer safety is typically realized by associating a lower and upper bound with each data pointer, and adding a check at runtime that verifies that memory accesses via the pointer fall within those bounds. Numerous pointer safety mechanisms based on such pointer bounds checks have been proposed. SoftBound [54] and HardBound [55] perform pointer bounds checks against metadata stored in a shadow memory area. The bounds information for each pointer must be frequently retrieved from the shadow memory. SoftBound adds software checks to applications hardened with it, but breaks cache locality when retrieving pointer bounds. As a result, it leads to additional cache misses which hurt program performance. SoftBound incurs an average performance overhead of 67% in standard benchmarks. HardBound is a hardware-assisted scheme where the processor checks associated pointer bounds implicitly when a pointer is dereferenced. As the check is performed by hardware logic, the average performance overhead is reduced to ~10%. Both schemes have a worst-case memory overhead of ~200%.

Intel’s Memory Protection Extensions (MPX) is an Instruction Set Architecture (ISA) extension for pointer safety introduced to Intel x86-64 processors in the late 2015 Skylake microarchitecture. MPX adds four new 128-bit registers for storing upper and lower pointer bounds and new instructions for managing the bounds registers and performing bounds checks on pointers. Bounds checks using the bounds registers are highly efficient. But since the number of bounds registers is limited, bounds information is also stored in tables with an index derived from the pointer address, similar to a two-level page table structure in x86. A 2GB intermediate table (bounds directory) is used as a mediator to the actual 4MB-sized bounds tables, which are allocated on-demand by the OS when bounds are created. The hardware performs a table walk of the bounds directory and bounds tables when bounds information is fetched to the registers. Oleksenko et al. [56] found that MPX
incurs an average performance overhead of 50% and a memory overhead of ~90%, largely due to
the complexity of storing and loading bounds metadata.

*Fat-pointer* schemes store the associated bounds metadata [57] together with pointers, e.g., by in-
creasing their length [58] or by borrowing unused bits from pointers [57]. Re-purposing parts of
a pointer to store validation data has the advantage of enabling fast retrieval of pointer metadata
without a need for lookups from disjoint memory. But it changes the representation of pointers in
memory in ways that break both binary and source code compatibility. Fat-pointers have primar-
ily been deployed in clean-slate ISA designs [59], and memory-safe programming languages, e.g.,
Cyclone [60] and Rust [61]. BIMA [59] is a hardware-assisted fat-pointer scheme for the SAFE
secure computing platform [62]. BIMA limits the virtual addresses to 46 bits and restricts pointer
alignment to powers of two. This frees 18 bits in 64-bit pointers for encoding bounds information.
BIMA demonstrates that on a clean-slate ISA design, fat pointers can be realized without a perfor-
ance penalty, and a 3% memory overhead due to segmentation caused by alignment restrictions
on BIMA pointers.

*Low-fat-pointers* [63,64] are an alternative to fat pointers compatible with commodity 64-bit hard-
ware architectures, such as x86-64. Low-fat-pointers require customized stack and heap allocators
that restrict both stack frame and heap memory allocation sizes to a fixed finite set, and split the
main program stack and heap into several sub-stacks and sub-heaps, one for each possible alloca-
tion size. Pointer accesses are then validated according to the allocation bounds associated with the
corresponding sub-stack or sub-heap. The improved compatibility comes at the cost of accuracy,
as low-fat-pointers accesses are only enforced at allocation bounds. On average, low-fat-pointers
adds a performance penalty of 54% (16% for out-of-bounds writes) and memory overhead of 15%
for stack data, and incurs a 56% performance (13% for out-of-bounds writes) and 11% memory
overhead for heap data.

*Data integrity* [65,66] prevents data manipulation by protecting against invalid/unintentional mem-
ory writes, but with reads left unchecked. It is an approximation of the spatial memory safety (e.g.,
preventing out-of-bound writes) in terms that data integrity typically maintains bounds based on
static points-to analysis. Before each write dereference, it checks whether the location is within its
valid points-to memory region. Since it only enforces the spatial memory integrity, and thus cannot
prevent attacks exploiting use-after-free and double-free vulnerabilities.

**Pointer Authenticity/Integrity**

*Pointer authenticity/integrity* aims to ensure the validity of pointers, *i.e.*, the value of a pointer (the
address of the target object) is not arbitrarily controllable by an attacker, even in the presence of
memory corruption vulnerabilities that may allow a manipulation over the pointer value.
**PointGuard** [67] encrypts all pointers at runtime by XORing them against a key generated at program initialization. The encryption on each pointer must be reversed before dereferencing a pointer. PointGuard incurs a small to medium overhead (0%~20%), but is vulnerable to information disclosure, e.g., if an attacker learns the key or the XORed ciphertext of a pointer to a known address.

**ARMv8.3 Pointer Authentication (PA)** [68] is a hardware pointer authenticity primitive introduced in the ARMv8-A processor architecture. PA introduces a set of new instructions for calculating and verifying a *Pointer Authentication Code* (PAC) for pointers. The PAC is a keyed MAC of the pointer value and a 64-bit context, which limits the area of validity of the pointer, e.g., by binding it with the current value of the stack pointer. The PAC is stored in the unused bits of 64-bit pointers and verified before the pointer is dereferenced. The ARMv8.3 architecture provides four keys for PAC (two for code pointers and two for data pointers) and a fifth key intended for authenticating data. The keys are stored in internal CPU registers which are not accessible from userspace code.

**Code-Pointer Integrity (CPI)** [69] provides control-flow hijacking protection rather than the complete memory safety. Therefore, it incurs a very low performance overhead with around 1.9% (C program) or 8.4% (C/C++ program) slowdown. Kuznetsov et al. [69] also introduced a relaxation of CPI with better performance properties, called code-pointer separation (CPS), to achieve better security-to-overhead trade-off.

However, pointer-based approaches generally suffer from a poor scalability in terms of increased execution time and memory consumption as the number of protected pointers increases. They also require a comprehensive understanding of a program’s memory layout at individual pointer granularity over time in order to differentiate between benign (within bounds) memory accesses from malicious (out-of-bounds) memory accesses. Another concern is the compatibility problem with unprotected modules, which may cause false alarms when a pointer is modified by an unprotected module.

**Object-Based Approaches**

Instead of enforcing bounds checking with pointers, object-based approaches detect out-of-bounds memory accesses to objects. It solves the compatibility issues caused by pointer based approaches.

**AddressSanitizer (ASan)** [27] is a memory error detector for Linux available in GCC and Clang/L-LVM. It can detect out-of-bounds memory accesses to global, stack, and heap objects. In addition, it can detect a number of *temporal memory errors*, such as use-after-free and double free conditions. ASan tracks objects stored in application memory by storing metadata on each object in a disjoint *shadow memory* area that occupies a fraction of the application’s virtual memory space. The shadow memory records which memory regions in the application memory are allocated and used,
and therefore safe to access. However, these memory safety guarantees do not preclude erroneous memory accesses in which a corrupted pointer de-references a valid, but unintended memory object. In addition, ASan places blocks of "poisoned" memory between adjacent objects in the stack, heap and global storage (i.e., blacklisting unsafe memory regions). Different from approaches that whitelist safe memory regions, poisoned memory is marked as invalid in the application’s shadow memory, and acts as "red-zones", which, if accessed, indicates a contiguous overflow, e.g., beyond the array boundary. However, memory errors that enable non-contiguous accesses or accesses with a larger step distance than the size of the red-zone can violate spatial safety without setting off the tripwire. Hardware-assisted AddressSanitizer (HWASAN) [70] is a tool similar to AddressSanitizer, but based on partial hardware assistance. It relies on address tagging support which is only available on ARM’s 64-bit architecture (AArch64).

2.2.2 S2 Defense

S2 defenses aim at limiting the consequences of attacks that leverage memory vulnerabilities, including software compartmentalization and randomization (diversification) techniques.

Software Compartmentalization

Software Fault Isolation (SFI) [49] compartmentalizes software in a single address space by sandboxing distrusted modules into separate fault domains, which are arranged to occupy a distinct portion of the program’s address space. SFI-enforcement ensures code in the fault domain is unable to directly access memory or jump to code outside the reserved portion of address space. It can only interact with code outside its domain through well-defined call interfaces.

XFI [50] is a SFI variant for Microsoft Windows for isolating shared libraries within an application, or drivers within the kernel. However, XFI does not protect against confused deputy attacks, where a distrusted module abuses an over-permissive kernel routine that the module is allowed to invoke. LXFI [51] extends SFI to Linux kernel modules, which exhibits a more complex interface, e.g., callbacks invoked by the kernel make manual interposition more difficult. LXFI also enables compartmentalization between different instances of a single module, e.g., a kernel driver which may instantiate server module principals, such as block devices or sockets. CHERI [72] is a hardware-assisted capability model for the 64-bit MIPS ISA that can support different protection models, such as pointer safety and software compartmentalization [73, 74].
Randomization

Randomization aims to hide attack targets by randomizing the location of program segments [75], layout of the code [76], layout of the data [52] or the data itself [77] so that unauthorized access would lead to unpredictable behavior. In particular, data space randomization [52, 78, 79] aims to randomize the representation of data stored in program memory at runtime to make it unpredictable for unauthorized accesses, and thus reducing the possibility that attackers can leak security-critical memory addresses or manipulate the content of the targeted data.

Though randomization can be more efficient to maintain, these probabilistic approaches rely on some secrecy assumptions (e.g., a XOR mask or randomization secret), which makes them susceptible to information leakage attacks.

2.2.3 Defenses Specific to Data-Oriented Attacks

In addition to generic memory corruption prevention mechanisms, a number of detection and prevention techniques specially focusing on data-oriented attacks have been proposed in the literature. In this section, we discuss these defense mechanisms.

<table>
<thead>
<tr>
<th>Defense and Year</th>
<th>Stage</th>
<th>Approach</th>
<th>Security Guarantee</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>YARRA [18], 2011</td>
<td>S1 (Pointer safety)</td>
<td>Program instrumentation</td>
<td>User-specified critical data</td>
<td>400%~600% (whole program)</td>
</tr>
<tr>
<td>HardScope [22], 2018</td>
<td>S2 &amp; S3 (Compartmentalization)</td>
<td>Hardware extension</td>
<td>Context-specific memory isolation</td>
<td>~3.2%</td>
</tr>
<tr>
<td>PrivWatcher [80], 2017</td>
<td>S2 (Compartmentalization)</td>
<td>Kernel modification</td>
<td>Protect process credentials data in Linux kernel</td>
<td>~3% (95% in extreme cases)</td>
</tr>
<tr>
<td>HDFI [53], 2016</td>
<td>S2 (Compartmentalization)</td>
<td>Hardware extension</td>
<td>Data-flow isolation</td>
<td>~2%</td>
</tr>
<tr>
<td>PT-Rand [25], 2017</td>
<td>S2 (Randomization)</td>
<td>Kernel modification</td>
<td>Protect kernel page tables</td>
<td>~0.22%</td>
</tr>
<tr>
<td>DFI [28], 2006</td>
<td>S3</td>
<td>Program instrumentation</td>
<td>data-flow integrity</td>
<td>~100%</td>
</tr>
<tr>
<td>CVI [81], 2018</td>
<td>S3</td>
<td>Program instrumentation</td>
<td>Data-flow integrity</td>
<td>~2.7%</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of defensive mechanisms against data-oriented attacks

Data-Flow Integrity (DFI) [28] ensures that whenever a value is read from memory, the instruction that wrote the value to memory is in the set of reaching definitions for the read, i.e., an earlier instruction that assigns the variable a value without an intervening assignment. The reaching definitions are determined at compile-time through static analysis that produces the program’s data-flow graph. The program is instrumented to record the last instruction that wrote to each variable. On
each read instruction, the origin of the last write instruction is checked against the pre-computed data-flow graph. DFI can be instantiated at different granularities. **Intra-procedural DFI** only instruments uses of control-data and uses of local variables without definitions outside their function. **Inter-procedural DFI** isolates individual data-flows from each other. Intra-procedural DFI incurs 44% and Inter-procedural DFI incurs 103% runtime performance overhead, respectively, and approximately 50% space overhead (for instrumentation) [28].

**YARRA** [18] is a C language extension that validates a pointer’s type for **critical data types** annotated by the programmer, which is of an S1 defense. Similar to DFI [28], YARRA guarantees that critical data types are only written through pointers with the given static type. YARRA is suitable for hardening access to isolated pieces of critical data, such as cryptographic keys stored in program memory at runtime. However, when applied for whole program protection, it incurs a performance overhead in the order of 400%~600%. In addition, YARRA protection relies on programmers’ manual annotations, and thus can be error-prone especially for complicated programs.

**HardScope** [22] is a runtime variable scope enforcement to mitigate data-oriented attacks by introducing the intra-program memory isolation. On each memory access (i.e., load/store), HardScope enforces that the memory address requested is in the accessible memory areas. Nyman et al. [22] demonstrated the effectiveness of HardScope for the new RISC-V (reduced instruction set computer) open instruction set architecture, by introducing a set of six new instructions. HardScope instructions are instrumented at compile-time, and memory access constraints are enforced at runtime. It shows that HardScope has a real-world performance overhead of 3.2% in embedded benchmarks. The authors highlighted that though HardScope significantly reduces the number of available DOP gadgets, it is still not possible to defeat all DOP attacks in arbitrary programs.

**PrivWatcher** [80] is a framework particularly for monitoring and protecting the integrity of process credentials (i.e., task_struct that describes privileges of a process in Linux kernel) against non-control data attacks. It involves a set of kernel modifications including relocating process credentials into a safe region, code instrumentation and runtime data integrity verification, in order to provide non-bypassable integrity assurances. It ensures the Time of Check To Time of Use (TOCTTOU) consistency between verification and usage contexts for process credentials by adopting a dual reference monitor model. Authors implemented the PrivWatcher prototype in Ubuntu Linux running on x86-64. The experiment results show that PrivWatcher incurs a overhead less than 3%. But it incurs more than 94% overhead for applications that involve installing new task_struct structures to processes.

**Hardware-Assisted Data-flow Isolation (HDFI)** [53] extends the RISC-V ISA to provide instruction-level granularity isolation by tagging each machine word in memory and every memory access instruction with a protection domain. However, unlike software-enforced DFI, HDFI only supports two simultaneous protection domains. Tagged memory architectures, e.g., **lowRISC** [82], in general
can be used to implement sophisticated compartmentalization strategies. *Intel Memory Protection Keys* (MPK) [83] provides a hardware support for page granularity tagged memory. MPK allows a process to assign memory pages to one of 16 distinct protection domains and selectively enables or disables access to each region independently.

Davi *et al.* [25] presented a data-oriented attack against kernel page tables to bypass the CFI-based kernel hardening technique, and subsequently attackers can execute arbitrary code with kernel privileges. To mitigate the threat, they proposed PT-Rand, which randomizes the location of page tables to prevent attackers from manipulating page tables by means of data-oriented attacks. Evaluation results show that PT-Rand on Debian only incurs a low overhead of 0.22% for common benchmarks. However, it is still possible that attackers undermine these schemes if the secret information (e.g., randomization secret) is leaked or inferred [22].

**CVI** (Critical Variable Integrity) [81] verifies define-use consistency of critical variables for embedded devices. The define-use consistency is defined as the value of a variable cannot change between two adjacent define- and use-sites. After identifying critical variables (either automatically identified or manually annotated), the compiler inserts instrumentation at all the define- and use-sites for these critical variables, to collect values at runtime and send them to an external measurement engine. CVI checking compares the current value of a variable at every use-site, and the recorded value at the last legitimate define-site. However, like DFI [28], CVI is based on compile-time instrumentation and frequent runtime checking, which incurs a high overhead if a large number of variables are protected.

Table 2.1 compares representative data-oriented attack specific defense mechanisms. Most approaches are recently proposed except DFI [28]. However, none of these solutions is perfect to defend against data-oriented attacks. PrivWatcher [80], HDFI [53], PT-Rand [25], and CVI [81] protect specific non-control data. They cannot be directly applied to protect user-space applications against general data-oriented attacks, in particular the DOP attacks. DFI [28] and YARRA [18] incur a high performance overhead. HardScope [22] cannot completely defeat all DOP attacks.

### 2.3 Program Anomaly Detection

Program anomaly detection has been an effective approach in detecting zero day attacks, which enables proactive defense against new and unknown attacks. It implements an application-level reference monitor by modeling normal program behavior. Such monitoring and detection approach possesses two favourable properties: 1) It requires no modification of the binary code and thus supports legacy programs/applications; and 2) It detects a wide range of attacks/errors, e.g., not only memory corruption attacks, but also hardware/logic errors and misconfigurations, etc. Typically
Figure 2.2: Training phase and monitoring phase in program anomaly detection

Program anomaly detection is composed of two stages, as shown in Fig. 2.2. During a training phase, the model of benign program behavior is constructed, e.g., using machine learning techniques. In the monitoring phase, an anomaly detector observes the program execution. Any deviations from the normal model are identified as anomalies. Runtime attestation \[32, 84\] is similar to program anomaly detection in that both approaches rely on models of the normal program behavior, and detect abnormal behaviors that deviate from the model at runtime. The difference lies in that runtime attestation collects/monitors execution behaviors in an on-demand manner, e.g., a trusted party (i.e., verifier) randomly sends a request to verify the integrity (such as the CFI) of a running program (i.e., prover).

The primary task of program anomaly detection is to model a program’s expected normal behavior (i.e., program behavior modeling), so that an anomaly detector can recognize potential attacks by their deviations for normal behaviors with low false alarm rates. Program tracing is an important supporting technique to enable program anomaly detection, which involves recording information about a program’s execution. It determines the practical deployability and acceptance of a program anomaly detection system. In this section, we discuss these two important aspects.

### 2.3.1 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 \[39\]. Existing research efforts mainly focus on proposing different models to accurately describe normal application behaviors. We introduce representative modeling techniques for program anomaly detection.

Warrender et al. \[41\] presented the comparison of four different program behavior models, including simple enumeration of sequences (e.g., n-gram), sequence frequency-based (i.e., n-gram with
frequency threshold, named t-stide in [41]), rule induction-based data mining approach, and Hidden Markov Model (HMM). System-call data is used to characterize the normal behavior of programs in all these models. The assumption is that a compromised application cannot cause much harm unless it interacts with the underlying OS, and the only way to interact with an OS is normally via system-calls. Evidence shows that short sequences of system-calls (i.e., n-grams and variants) are a good discriminator between normal and abnormal executions of many programs. Wagner et al. [85] proposed to model program behavior for intrusion detection using static analysis from program source code, where the authors built a non-deterministic finite state automaton (FSA) and a pushdown automaton (PDA) to characterize the expected system-call traces, respectively. Giffin et al. [86] pointed out that PDA models suffer the stack state explosion issue, and thus are prohibitively expensive to operate. Unlike Wagner’s approach that learns the automaton via static analysis, Sekar et al. [87] proposed to construct an FSA via dynamic learning from past traces. FSA-based models are subject to the impossible path problem (i.e., nondeterminism ambiguity) [88], since it cannot capture the function call context and thus allows impossible paths.

To mitigate non-determinism and acceptance of illegal sequences of calls, Liu et al. [88] extended the FSA and PDA models introduced in [85], and presented a hybrid push down automata model (HPDA), using a combination of static analysis and dynamic learning. The basic idea is to use static analysis to obtain a base model and then to use dynamic learning as a supplement to capture behaviors that are missed in static analysis. Different from the PDA model in [85], the HPDA utilizes the call stack information maintained by the system rather than maintaining an extra stack. VPStatic [89] and VtPath [90] are two deterministic pushdown automaton (DPDA) models taking advantage of call stack information to remove nondeterminism ambiguity. Giffin et al. proposed the Dyck model in [86], a statically constructed context-sensitive program representation by inserting precall and postcall at each call site. Dyck model is as powerful and expressive as the full PDA model, but operates with a overhead close to the context-insensitive model.

Since early program behavior models mentioned above have limited detection capability [39], several recent models have been proposed. Xu et al. [91] proposed a probabilistic HMM-based control-flow model representing the expected call sequences of the program for anomaly detection. Probabilistic modeling is capable of providing the maximum likelihood associated with a call sequence occurrence, thus can detect aberrant call frequencies, e.g., service abuse attacks. Shu et al. [40] proposed an anomaly detection approach with two-stage machine learning algorithms for long-range program behavior modeling. The basic idea is to mine co-occurrence of system events (i.e., function/system-calls) in a long-range execution window to detect subtle program inconsistencies and anomalies.

Though various program behavior models have been proposed in the past decade [39], these approaches mainly focus on detecting control-flow attacks. The frequency and co-occurrence-based
anomaly detection approaches [40, 42] model program behavior at the system-call or function-call level. These coarse-grained models capture incomplete control-flow behaviors, and thus can only detect a limited number of data-oriented attacks.

### 2.3.2 Software-Based Program Tracing

The main barrier hindering the wide deployment of program anomaly detection is the monitoring/tracing overhead [92]. This is because, the program monitoring/tracing in most existing anomaly detection systems is purely software-based. Traditional software-based tracing can be very expensive. It suffers from a large performance degradation due to constant monitoring operations, which usually involve interactions between the monitored process and operating system.

We experimentally compare the tracing overhead of typical system-call tracing tools including PIN, SystemTap, and Strace. These tools are commonly used for program anomaly detection [40], system diagnosis [93], and malware analysis [94]. We use three utility applications (i.e., tcas (1608 test cases), replace (5472 test cases), schedule (2650 test cases)) from the Software-artifact Infrastructure Repository (SIR) benchmark suite [95]. We measure the runtime performance overhead of these programs running on a desktop computer (Ubuntu 16.04, Intel Xeon processor 3.50GHz and 16GB of RAM). Table 2.2 shows the results. The baseline refers to the execution time without tracing. SystemTap’s instrumentation incurs a relatively low overhead with around 30% slowdown (but it requires a long instrumentation time before the program execution). The runtime performance overhead of Strace shows around 2.5x slowdown. When tracing the call stacks on every system-call, it yields around 4x slowdown. The PIN tool is based on the dynamic binary instrumentation. Since we insert a checkpoint at every instruction to check whether the current instruction is a system-call invocation, it incurs a rather significant runtime overhead.

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>PIN</th>
<th>SystemTap</th>
<th>Strace</th>
<th>Strace w/ callstack</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>0.000018s</td>
<td>0.024063s</td>
<td>0.000023s</td>
<td>0.000046s</td>
<td>0.000079s (4.49x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1336x)</td>
<td>(1.28x)</td>
<td>(2.56x)</td>
<td></td>
</tr>
<tr>
<td>replace</td>
<td>0.000066s</td>
<td>0.051280s</td>
<td>0.000079s</td>
<td>0.000155s</td>
<td>0.000256s (3.87x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(776x)</td>
<td>(1.20x)</td>
<td>(2.35x)</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td>0.000073s</td>
<td>0.107846s</td>
<td>0.000095s</td>
<td>0.000164s</td>
<td>0.000258s (3.53x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(1477x)</td>
<td>(1.30x)</td>
<td>(2.25x)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Tracing overhead of different software-based system-call tracing tools
2.3.3 Hardware-assisted Program Tracing

To overcome the limitation of software-based tracing, Zhang et al. [96] proposed a hardware-based approach to monitor the control-flow of a program to detect anomalous execution. As hardware tracing infrastructures are increasingly embedded in modern processors (e.g., Intel’s Processor Trace and ARM’s CoreSight), hardware-assisted anomaly detection has received much attention. The hardware tracing capability brings the anomaly detection deployment closer to reality. In this section, we introduce the Intel’s Processor Trace (PT), which is used for tracing control-flow transfers in our program anomaly detection systems in Chapters 4 and 5.

Intel Processor Trace (PT)

PT tracing is a commercially available low-overhead hardware feature on Intel CPUs, which enables the construction of complete control-flow after program execution completes. To capture control-flow information of program execution, PT records target addresses of indirect branches (i.e., TIP packets/events) and taken/non-taken decisions of conditional branches (i.e., TNT packets/events). TIP.PGE (Packet Generation Enable) and TIP.PGD (Packet Generation Disable) denote the beginning and the end of tracing, respectively. The trace format in PT is highly compressed/encoded to achieve an efficient logging. For example, it uses one bit to indicate taken or not-taken for a conditional branch in a TNT packet. As a result, PT tracing only incurs less than 5% overhead on average [43]. Though PT was originally designed for detailed offline debugging and failure diagnosis, recent research has shown its applicability for online security enforcement to defend against control-flow attacks [43, 44, 45, 46].

To the best of our knowledge, the applicability of using PT for detecting data-oriented attacks has not been addressed in the literature. With the increasing availability of hardware-assisted tracing infrastructures, PT-based program anomaly detection has the potential to be the last line of defense against data-oriented attacks.

PT for Security

More recently, several research efforts explore the effectiveness of PT for CFI enforcement [43, 44, 45, 46]. Software-only CFI implementations have several limitations, such as coarse-grained enforcement (e.g., static binary instrumentation), lack of flexibility (e.g., compile-time instrumentation) or high performance overhead (e.g., dynamic binary instrumentation) [43]. Hardware-assisted CFI enforcement avoids these limitations by using hardware-generated events to check indirect control-flow transfers, e.g., using the Last Branch Record (LBR) [97], Branch Trace Store (BTS)
and Performance Monitoring Unit (PMU) \cite{98} features to enforce CFI. PT is a successor to LBR and BTS, but provides several advantages over them. It supports unlimited path history and incurs a very low overhead.

GRIFFIN \cite{43} is an operating system mechanism (running in kernel) that leverages the Intel PT feature for enforcement of CFI policies. It focuses on the effective online enforcement and assumes that CFI policies are given. PT-CFI \cite{44} is a backward-edge control-flow violation detection system using PT tracing, including offline training and online deep inspection phases. The detection is based on the TIP Graph (TIP-G), which is constructed in the training phase and composed of indirect control-flow transfers (i.e., call, ret, jmp). At runtime, unknown TIP events trigger the deep inspection, which is used to disassemble the corresponding binary code and determine whether the TIP event is legal or not. FlowGuard \cite{45} constructs the PT-compatible indirect targets connected control-flow graph (ITC-CFG) to detect control-flow violation. Through static binary analysis, it first constructs a conservative CFG (i.e., over-approximation of indirect targets). Then, a training phase labels selective edges with credits and additional branch taking information, where an edge will be assigned a high-credit if this edge exists during the training phase. At runtime, FlowGuard performs slow path checking for low credit edges, which parses the binaries and combines the traced packets for the entire control-flow path to check whether it is a valid control-flow transfer. To address the over-approximation problem of control targets in forward-edge CFI, PITTYPAT \cite{46} utilizes PT to track basic blocks being executed to compute the reduced legal control transfer target sets through runtime path-sensitive point-to analysis. To enable online CFI policy checking, all these approaches use critical system-calls as checkpoints, i.e., enforcing the CFI policy at the entry points of selected system-calls. Though they can effectively defend against ROP-like control-flow hijacking attacks, none of the existing work investigates the possibility of using PT to defeat data-oriented attacks.

Understanding the Detectability by PT-based Detection Against Data-Oriented Attacks

We discuss the limits of protection offered by PT-based program anomaly detection. Detecting data-oriented attacks via PT tracing requires that an attack manifests unusual control-flow behavior, either incompatible branch behaviors or frequency anomalies. Typically, uses of non-control data in a program can be classified either as predicate uses or non-predicate uses (such as computation uses). A predicate-use directly affects the control-flow behaviors. While a non-predicate use may affect the computation or the output of a program \cite{99}. Zhuang et al. \cite{100} showed that many non-control data attacks actually cause changes in control-flows. In their experiments, on average 49.4% of memory tamperings change program control-flows.

When a manipulated variable is only used for computation or output (i.e., non-predicate use), and
the exploit does not incur any side effect on control-flow transfers, such an attack is undetectable by PT-base control-flow tracing. We discuss typical undetectable cases, which are mainly direct data manipulation (DDM) attacks.

- Corrupting user identity for privilege escalation: Simply corrupting a user identify data (e.g., UID) may lead to a compromise of the root privilege. However, for an undetectable data manipulation with the privilege escalation, an attacker usually goes after malicious actions once obtaining the new privilege, e.g., launching a shell. Such malicious actions can be easily detected by CFI-based detection.

- Corrupting configuration data: It is likely that corrupting configuration data via format string vulnerabilities can evade PT-based detection, since format string vulnerabilities allow a single memory write without any side effect on control-flow behavior.

- Constructing exfiltration channels for information leakage: Attackers exploit an existing information outlet (also known as sink such as printf or send functions) for information leakage by replacing the pointer value of the outlet function’s parameter with the address of the data to be exfiltrated. Such an attack may not incur any anomalous control-flow behavior.

2.4 Anomaly Detection in Cyber-Physical Systems

Due to the diversity of CPS applications, existing anomaly detection solutions are proposed to detect specific attacks for specific applications, such as smart infrastructures [29], unmanned aerial vehicles [101], medical devices [102], automotive [103, 104], industrial control process [105, 106, 107]. The majority of research efforts in this area thus far have concentrated on behavior model-based anomaly detection [107], 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). 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 [105].

In this thesis, we present a new program behavior model named eFSA in Chapter 3, which 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 2.3 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. [42] proposed a lightweight method for detecting anomalous executions using the distribution of system-call frequencies. The frequencies are
<table>
<thead>
<tr>
<th>Research Work</th>
<th>Category</th>
<th>Approach</th>
<th>Security Guarantee</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yoon et al. [42]</td>
<td>Program behavior model (cyber)</td>
<td>Syscall frequencies</td>
<td>Frequency-based program control-flow anomaly</td>
<td>Raspberry Pi testbed</td>
</tr>
<tr>
<td>Feng et al. [108]</td>
<td>Network traffic analysis (cyber)</td>
<td>Machine learning based traffic analysis</td>
<td>Traffic alteration</td>
<td>Traffic data from a gas pipeline system</td>
</tr>
<tr>
<td>Zimmer et al. [35]</td>
<td>Timing analysis model (cyber)</td>
<td>Static/dynamic timing analysis</td>
<td>Code injection attacks</td>
<td>Simulation/Testbed</td>
</tr>
<tr>
<td>C-FLAT [32]</td>
<td>Program behavior model (cyber)</td>
<td>Program analysis and instrumentation</td>
<td>Control-oriented attacks and limited non-control-data attacks</td>
<td>Raspberry Pi testbed</td>
</tr>
<tr>
<td>Hadziosmanovic et al. [109]</td>
<td>Range-based model (physical)</td>
<td>Attribute values extracted from network traffic</td>
<td>False data injection attacks</td>
<td>Traffic data from water treatment plants</td>
</tr>
<tr>
<td>Cardenas et al. [105]</td>
<td>Physical laws</td>
<td>Linear model derived from training data</td>
<td>False data injection attacks</td>
<td>Simulation</td>
</tr>
<tr>
<td>SRID [110]</td>
<td>Physical laws</td>
<td>Correlation analysis of system variables.</td>
<td>False data injection attacks</td>
<td>Simulation</td>
</tr>
<tr>
<td>C [111]</td>
<td>Control policies (physical)</td>
<td>User specified control policies</td>
<td>Control signal violation</td>
<td>Raspberry Pi testbed</td>
</tr>
<tr>
<td>eFSA (Chapter 3)</td>
<td>Cyber-physical model</td>
<td>Event-aware FSA</td>
<td>Data-oriented attacks</td>
<td>Raspberry Pi testbed</td>
</tr>
</tbody>
</table>

Table 2.3: Comparison of representative CPS anomaly detection approaches

for individual system-calls, *i.e.*, 1-grams. The authors in [112] proposed a hardware based approach for control-flow graph (CFG) validation in runtime embedded systems. McLaughlin et al. [106] 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. Aubel et al. [113] proposed a program anomaly detection system that uses electromagnetic side-channel measurements to detect behavioural changes of the software running on industrial control systems. It is intended to defend against an attacker who can upload new software to the PLC to replace or modify the existing benign program.

C-FLAT [32] instruments target control programs to achieve the remote attestation of execution paths of monitored programs. The normal execution of control-flow paths is derived 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 an application’s control-flow has been compromised. C-FLAT [32] is the most related work to our proposed eFSA (Chapter 3). Both C-FLAT and eFSA 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 from
the 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. [108] 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 baseline 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. [35] used the worst-case execution 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. [114] focused on detecting intrusions in real-time control systems. Yoon et al. [115] presented SecureCore, a multicore architecture using the timing distribution property of each code block to detect malicious activities in the real-time embedded system. Lu et al. [116] 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. [109] 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 from the 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. [110]
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. [103] 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 [109] and power grid [117], 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. [105] 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. [107] 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. [111] 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 eFSA 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 2.3, 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 [117]. 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 [32, 42] are effective against control-flow 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. Contex-IoT [118] provides context identification for sensitive actions in the permission granting process of
IoT applications on Android platforms. Though both ContextIoT and our proposed eFSA consider execution contextual integrity, ContextIoT does not support the detection of data-oriented attacks.

Distinctive from existing works in this area, my work in Chapter 3 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. The Stuxnet attack [33] manipulated the nuclear centrifuge’s rotor speed, and fooled the system operator by replaying the recorded normal data stream during the attack [38]. Since eFSA’s detection is independent on the history data, it makes Stuxnet-like attacks detectable in eFSA by detecting runtime inconsistencies between the physical context (runtime rotor speed) and the control program’s behavior. In addition, attackers may exploit hardware vulnerabilities [119] to manipulate data in memory so as to launch attacks on control branch or control intensity. eFSA is also able to detect anomalies caused by such hardware attacks.
Chapter 3

Event-Aware Program Anomaly Detection in Cyber-Physical Systems

3.1 Instruction

Cyber-physical systems are widely used to operate critical infrastructure assets, such as electric power grid, oil and natural gas distribution, industrial automation, medical devices, automobile systems, and air traffic control [29]. Recent studies [35, 36, 37, 32, 38, 22] have shown that control programs suffer from a variety of runtime software exploits. These attacks can be broadly classified into two categories: control-flow attacks and data-oriented attacks. Control-flow attacks in conventional cyber systems (i.e., without cyber-physical interactions) have been well studied [39]. It is possible that existing detection approaches [121, 12, 122, 112, 116, 3] are extended to defend against control-flow attacks in CPS. Data-oriented attacks manipulate program’s internal data variables without violating its Control-Flow Integrity (CFI), e.g., non-control data attacks [1], Data-Oriented Programming (DOP) [21]. Data-oriented attacks are much more stealthy than attacks against control-flows. Because existing CFI-based solutions are rendered defenseless under data-oriented attacks. We mainly focus on runtime software exploits, and thus sensor data spoofing attacks [110, 123] in the physical domain are out of the scope in this work.

Since many control decisions are made based on particular values of data variables in control pro-

---

1Though 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 the 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 [120].
grams [32], 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.

- **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 [124] or preventing a blast furnace from being shut down properly as in the recent German steel mill attack [125]).

- **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 [32]).

In many instances, CPS can be modeled as *event-driven* control systems [126, 118]. 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 CFI-based detections [21]. 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 [42, 32] and they are ineffective to prevent both attack types due to the lack of runtime execution semantics checking.

**Goals and Contributions.** In this work, 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 [124]. None of existing program anomaly detection solutions [39] has the event-aware detection capability. 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. The main technical contributions are summarized as follows.
We describe a new security methodology, named *Orpheus*, that leverages the event-driven nature in characterizing CPS control program behaviors. We present a general method for reasoning about cyber-physical execution semantics of a control program, including the event identification and dependence analysis.

As an instantiation of *Orpheus*, we present a new event-aware finite-state automaton (eFSA) model to detect anomalous control program behaviors particularly caused by data-oriented attacks in CPS. By enforcing runtime cyber-physical execution semantics, eFSA detects subtle data-oriented exploits when physical events 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 [41] or HMM model [91].

We implement a proof-of-concept prototype on Raspberry Pi platforms, which have emerged as popular devices for building CPS applications [106, 32, 127]. 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 increasing 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 eFSA’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 ~0.0001s to check each state transition in eFSA model, ~0.063s for the local event verification, and ~0.211s for the distributed event verification.

The focus of this work 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 Section 3.8, 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.

### 3.2 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.

3.2.1 CPS Background

Fig. 3.1 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 [38]. 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. 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.

Embedded devices (a.k.a. field devices) in CPS are situated in the field, where their operating systems are typically embedded Linux/Windows variants [128] or PLC firmware [38]. 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 [128].

### 3.2.2 Attack Model and Assumptions

In this work, 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 about how attackers gained entry into the devices and launch different attacks, but focus on uncovering abnormal program execution behaviors after that [116]. 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 [42]. 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 [32]. 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 [35].

- **Our focus.** We focus our investigation on runtime software exploits, and thus sensor data spoofing attacks in the physical domain [123] 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 [129]. 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.
3.2.3 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 with control-flow graphs (CFGs). We illustrate our new detection capabilities using a smart syringe pump as an example. 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. 3.2. 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. 3.2(a) may trigger control actions `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. 3.2(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 }
```

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

(a) (b)

Figure 3.2: Two examples of data-oriented software exploits in a real-world CPS application

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 [32], which is based

---

2https://hackaday.io/project/1838-open-syringe-pump
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 ap-
proaches (e.g., global trace analysis [40] and system-call frequency distribution (SCFD) [42]) can-
not detect such either type of attacks, as their analysis do not model runtime cyber-physical context
dependencies.

3.2.4 Definition of Events

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

- Binary events return either True or False, which are defined in terms of pre-specified
status changes of physical environments and provide notifications to the control program
(e.g., Push_Event or Pull_Event in Fig. 3.2). Such events are commonly pre-defined
and used in CPS/IoT’s trigger-action programming (“if, then”) model [130, 118].

- 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. 3.2. It
  is challenging to identify non-binary events since they are not explicitly declared in control
  programs.

3.2.5 Orpheus Framework Overview

Fig. 3.3 shows the workflow of Orpheus event-aware anomaly detection framework, which is com-
posed 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 ➊, Orpheus identifies both binary events
and non-binary events involved in the control program. In step ➋, it performs the program de-
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 ➋, Orpheus conducts the program behavior modeling, such as the HMM-based model [91],
n-gram model [41], or control-flow integrity [12], which we refer to as a basic program behavior
model in Orpheus. The next step ➋ is important. It augments the basic model with event constraints
and obtains the event-aware program behavior model.
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 in Chapter 3. For example, HMM-based models [91] can be enhanced with event checking on event-dependent state transitions. For the n-gram model [41], 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 [12, 127] can also be augmented with event checking before executing control tasks.

### 3.3 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.

#### 3.3.1 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 [47] 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\(^3\) in our static analysis.

According to the definition of a non-binary event in Section 3.2.1, 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 [47], 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\(^4\), while the data dependency graph is at the granularity of instructions. Then, we obtain all conditional branch instructions with loops, by searching 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 [132] (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.

A more specific example of our event identification is illustrated in Fig. 3.4 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 (1). We then locate a conditional branch instruction with a loop (2). Suppose this conditional branch is data dependent on a sensor-reading API (3). On its true branch, if we find any actuation API (4), we consider the loop as a non-binary event. Finally, we record the search results for the next event dependence analysis (5).

We also design a similar procedure for identifying non-pre-defined 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.

\(^3\)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 [131].

\(^4\)In program analysis, a basic block is a linear sequence of instructions containing no branches except at the very end.
Algorithm 1: Identifying non-binary events

1. **Input:** Program $P$; Sensor-reading API set $A_{\text{API}}^s$; Actuation API set $A_{\text{API}}^a$
2. **Output:** Non-binary-event set $E_{\text{nb}}$

3. $E_{\text{nb}} \leftarrow \emptyset$
4. $G_{\text{pdg}} = \text{ConstructPDG}(P)$ /*construct the program dependence graph*/;
5. LoopBrSet = getLoopBranchSet($P$) /*get all the conditional branch instructions with loops*/;
6. **for** $\text{BranchInst}=\text{getNextInst}(\text{LoopBrSet})$ **do**
7. $S_{\text{bdd}} = \text{BackwardDataDependence}(G_{\text{pdg}}, \text{BranchInst})$;
8. /*Backward data dependent statements on BranchInst*/;
9. $S_{\text{fcd}} = \text{ForwardControlDependence}(G_{\text{pdg}}, \text{BranchInst})$;
10. /*Forward control dependent statements on BranchInst*/;
11. if $(S_{\text{bdd}} \cap A_{\text{API}}^s \neq \emptyset \& S_{\text{fcd}} \cap A_{\text{API}}^a \neq \emptyset)$ then
12. $E_{\text{nb}} = E_{\text{nb}} \cup \text{Event}(\text{BranchInst}, S_{\text{bdd}}, S_{\text{fcd}})$;
13. end

---

Figure 3.4: An example of identifying non-binary events

3.3.2 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 $BB_{\text{eta}}$, 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 $\text{FindEventDependence}$ (Line 5), once we find a basic block on which $BB_{\text{cur}}$ 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 $E_b \cap E_{\text{tmp}} = \emptyset$ avoids potential loops when including new events into $E_b$. Then, we recursively search any upstream event that $BB_{\text{cur}}$ depends on (Line 12).

**Algorithm 2: Event dependence analysis for binary events**

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

Fig. 3.5 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 $[\overline{E_1} \land E_2]$. 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 [133], which identifies statements triggered by a particular event during multiple rounds of program executions. It is worth mentioning that our event identification and dependence analysis is a general approach for reasoning cyber-physical execution semantics, independent of specific program anomaly detection models.
3.4  *eFSA: an Instantiation of Orpheus*

In this section, we describe details about how to build the event-aware finite-state automaton (*i.e.*, eFSA) model, a system-call level FSA-based instantiation of the *Orpheus* framework. eFSA captures the event-driven feature of CPS programs to detect evasive attacks.

### 3.4.1 Formal Description of eFSA

We construct the finite-state automaton (FSA) [87] 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 [134, 39].

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. 3.6(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, $S_0 S_1 S_2 S_3 S_4 S_5 S_6$, $S_0 S_1 S_2 S_3 S_4 S_5 S_6$, and $S_0 S_1 S_2 S_3 S_4 S_5 S_6$, the learnt FSA model is shown in Fig. 3.6(b), where each node represents a state and each arc represents a state transition.
Our eFSA model extends FSA with external context constraints, where event-dependent state transitions in FSA are labeled with event constraints. We formally define the eFSA model as a six-tuple: $(S, \Sigma, s_0, F, E, \delta)$. $S$ is a finite set of states which are PC values, and $\Sigma$ 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. $\delta$ 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 eFSA.

### 3.4.2 From Event-Annotated CFG to eFSA

To construct an eFSA 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 Section 3.3.1 and 3.3.2) 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 [85, 135]. In eFSA, 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 eFSA model.

![Diagram](image)

Figure 3.7: An example of the eFSA model

Fig. 3.7 shows an example of eFSA model corresponding to the FSA example in Fig. 3.6, where an event dependent transition is labeled by "([System Call] 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. 3.7, we identify binary-event dependent state transitions $[S_1 S_2 | E_1]$, $[S_1 S_3 | E_1 \land E_2]$, and a non-binary-event dependent control intensity loop $[S_2 S_4 | \overline{E_1}]$. It also contains an implicit event dependent transition $[S_1 S_4 | (E_1 \land E_2)]$.

### 3.4.3 Security Policies in eFSA

eFSA expresses causal dependencies between physical events and program control-flows. By checking execution semantics (i.e., enforcing cyber-physical security policies) at runtime, eFSA 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 cyberspace (i.e.,
system-calls) and sensor values in physical space. eFSA then enforces the policy that the observed control intensity should be consistent with the corresponding sensor measurements.

### 3.4.4 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 [136] 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 [106]. 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 Section 3.3.1, 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 Section 3.7.4). 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.
3.5 EFSA-based Detection

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

3.5.1 Runtime Monitoring and Detection

Our anomaly detector traces system-calls as well as the corresponding PC values during the execution of a control program. As shown in Fig. 3.8, 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 EFSA-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 [6]) which violate control-flow integrity of the model.

- **Event-dependent state transition.** In case of an event dependent state transition according to the EFSA model, we first perform the above basic state-transition checking. More importantly, with the help of the event verification (discussed in Section 3.5.2), 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.

### 3.5.2 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 eFSA 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 [107]. For example, one may utilize fluid dynamics and electromagnetics as the basic laws to create prediction models for water system [109] and power grid [117]. 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.

### 3.6 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, eFSA 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 [106, 32, 127]. 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 field. During the training phase, we collect program traces on Raspberry Pi and perform the eFSA 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. 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.

![Example of using strace tool with stack unwinding support.](image)

Figure 3.9: 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. 3.9 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)\(^5\) compiler infrastructure, based

\(^5\)http://llvm.org/
on an open source static slicer which builds dependence graphs 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 the 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 and c-sense-hat. 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.

### 3.7 Experimental Validation

We conduct CPS case studies, and evaluate eFSA’s detection capability against runtime data-oriented attacks. Our experiments aim to answer the following questions:

- **What is the runtime performance overhead of eFSA (Section 3.7.2)?**
- **Whether eFSA is able to detect different data-oriented attacks (Sections 3.7.3 and 3.7.4)?**

#### 3.7.1 CPS Case Studies

**Solard.** 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,

---

6https://github.com/mchalupa/dg
7http://experix.sourceforge.net/
8https://github.com/davebm1/c-sense-hat
9https://github.com/mrpetrov/solarmanpi
CriticalTempsFound() is a pre-defined binary event in Solard. When the temperature is higher than a specified threshold, the event function returns True.

**SyringePump**\(^{10}\). It was developed as an embedded application for Arduino platform. Abera *et al.*[32] 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[32]. 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.

### 3.7.2 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 3.1 illustrates eFSA’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 3.1, Raspberry Pi takes a much longer time (more than 150 times) than the 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 eFSA is 34 and 65, respectively.

Next, we measure the performance overhead incurred by eFSA’s anomaly detector on Raspberry Pi. The system-call tracing overhead has no difference between FSA and eFSA, incurring 1.5x~2x overhead in our experiments. Table 3.2 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-

---

\(^{10}\)https://github.com/control-flow-attestation/c-flat
Event Dependence Analysis

<table>
<thead>
<tr>
<th></th>
<th>Desktop Computer</th>
<th>Raspberry Pi 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solard</td>
<td>0.745s</td>
<td>109.975s</td>
</tr>
<tr>
<td>SyringePump</td>
<td>0.0035s</td>
<td>1.726s</td>
</tr>
</tbody>
</table>

Table 3.1: Average delay overhead in training phase

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.

<table>
<thead>
<tr>
<th>Delay (Raspberry Pi 2)</th>
<th>Mean</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSA State Transition Checking</td>
<td>0.00013293s</td>
<td>0.00004684s</td>
</tr>
<tr>
<td>Local Event Verification</td>
<td>0.06279120s</td>
<td>0.00236999s</td>
</tr>
<tr>
<td>Distributed Event Verification</td>
<td>0.21152867s</td>
<td>0.03828739s</td>
</tr>
</tbody>
</table>

Table 3.2: Runtime overhead in the monitoring phase

### 3.7.3 Detecting Attacks on Control Branch

In this experiment, we evaluate eFSA’s security guarantees against control branch attacks.

**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 [125], which may result in a blast furnace explosion. In this experiment, we attach the Raspberry Pi on an electric kettle (i.e., 1-Liter water boiler). The control program keeps monitoring temperature values. When the temperature is lower than 50°C, it turns on the heater. And when the temperature is higher than 60°C, where \texttt{CriticalTempsFound()} is supposed to return \texttt{True}, it turns off the heater. In the monitoring phase, when we detect an event-dependent state transition in eFSA model, the local event verifier performs event consistency checking.
Fig. 3.10 illustrates an instance of the Solard experiment. We corrupt the temperature sensor values in the range of 40~45°C, which falsifies the return value of `CriticalTempsFound()` to be always `False`. In every scan cycle, `eFSA` 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. 3.10. However, `eFSA` 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. 3.10. 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 [137] can be applied to enhance the detection accuracy.

**SyringePump**

In SyringePump, we set the threshold to 40$rH$, i.e., when the relative humidity value is higher than 40$rH$, 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. 3.11 illustrates an example of the experiment. The remote user command corrupts the humidity sensor value to be 48.56 rH, which falsifies the return value of `event-push` to be True. In case of any event-driven state transition according to `eFSA`, the event verifier checks consistency between the runtime execution semantics (e.g., the instantaneous humidity value) and program internal state. As shown in Fig. 3.10, `eFSA` raises an alarm when it finds a mismatch between the execution semantics and program behavior.

3.7.4 Detecting Attacks on Control Intensity

In this experiment, we demonstrate that `eFSA` 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 30 rH. The corrupted humidity value determines the amount of liquid to be dispensed, which equals to the humidity value subtracted by 30 rH 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.
Fig. 3.12(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 38$rH$. 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 20$rH$ and 48$rH$, 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. 3.12(b). In this test, if the difference between the derived value and the sampled average value from event verifier is larger than 3$rH$, 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, eFSA successfully detects the control intensity attacks.

From Section 3.7.3 and Section 3.7.4, 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, eFSA is able to detect potential anomalies.

### 3.8 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 [138]), 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 [139, 32]. 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 [128, 127]. For example, Raspberry Pi devices with embedded Linux OS have been used as field devices in many CPS/IoT applications [140]. Linux-based PLCs for industrial control have emerged to replace traditional PLCs [141] for deterministic logic execution. However, embedded devices may still operate in bare-metal mode [32], 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 [90]. 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 [40]. 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 [142]. The recent work, Ninja [143], 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.

**Generalization of eFSA:** Control programs running on embedded devices may receive network events from the control center, and then execute actuation tasks. Though eFSA 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 eFSA model is augmented with network events. Such an eFSA 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 [93].

### 3.9 Summary

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 eFSA, which advances the state-of-the-art program behavior modeling. To the best of our knowledge, this is the first program behavior model that integrates both cyber and physical properties to defend against data-oriented attacks. We implemented a proof-of-concept prototype to demonstrate the feasibility of our approach. Real-world case studies demonstrated eFSA’s efficacy against different data-oriented attacks. We also discussed limitations of our design and implementation, which leads to the work in the next chapter, such as extending the *Orpheus* design paradigm to support actuation integrity for fine-grained anomaly detection at the instruction level without the need of tracing facilities.
Chapter 4

Statistical Program Behavior Modeling for Frequency Anomaly Detection

4.1 Introduction

Frequency anomaly refers to an anomalous program behavior with aberrant occurrence frequencies of system events (e.g., system-calls, library-calls, function-calls, or control-transfers) [40]. Such anomalous patterns are the direct consequences of many data-oriented attacks. For example, corrupting non-control-data variables which directly or indirectly affect the amount of loop iterations can lead to abnormal patterns of the loop usage. Control-flow bending (CFB) attacks [3] and resource wastage attacks [16] can also lead to abnormal control-flow frequencies. On the other hand, despite a plethora of research on program anomaly detection [41, 85, 87, 144, 89, 90, 129, 145, 12, 42, 40, 91], attackers often attempt to evade or bypass the detection [146, 147]. It is desirable to raise the security bar for more accurate and precise program anomaly detection systems due to the increasing stealth in modern exploits.

We classify existing program anomaly detection approaches into two categories: 1) local anomaly detection (e.g., n-gram or automaton-based models [41]); and 2) long-range anomaly detection (e.g., distribution-based [42] or co-occurrence-based [40] models). Among these models, system-call based monitoring is widely used for detecting compromised programs, in comparison to library-calls or function-calls.

- *Local anomaly detection* inspects short-range segments of program execution traces to detect anomalies such as control-flow violations. N-gram based model defines the normal program behavior for a process by using short sequences of system-calls. Although short-range order-
ing of system-calls have a high probability of being perturbed when abnormal activities occur, it is vulnerable to mimicry attacks [146], which can be achieved by code injection exploits. An attacker may insert a malicious code, issuing system-calls accepted by a normal behavior model yet still carries out the same malicious action. Instead of using short sequences and being limited by length, automaton/state-based models use finite state automaton (FSA) or hidden Markov model (HMM) to express every possible sequence. Using program counter (PC) information (i.e., return addresses) in FSA/HMM significantly increases the resistance against control-flow attacks. However, frequency anomaly does not violate CFI and may not introduce any unknown short call sequences (i.e., n-grams). Therefore, such detection models that capture local execution context features cannot efficiently detect frequency anomalies.

• **Long-range anomaly detection** examines longer system behaviors (e.g., a complete program behavior instance) than the local anomaly detection. Co-occurrence-based models are recently proposed to defend against these stealthy attacks, which improve the simple enumeration in local models by considering the frequency distributions of system-call events (e.g., histogram of individual system-call frequency [42]). However, such models neglect the temporal relations among system-calls (single/pair frequency analysis is inadequate). Since attackers often attempt to evade or bypass the deployed detection, such long-range models provide insufficient security against mimicry attacks.

In this work, we present a statistical program behavior modeling framework, which combines the local and long-range models to improve the robustness against mimicry attacks and significantly increase the difficulties that an attack bypasses the anomaly detection system. The key idea is to model normal behaviors of a program by frequency distributions of n-step (e.g., n=2,3,4) program execution sequences on top of underlying local models such as n-gram or FSA models. Then, cluster analysis is used to learn distinct normal program behaviors from a set of n-step frequency distributions.

The main contributions of this work are summarized as follows:

• We present a general framework for fine-grained program anomaly detection by characterizing statistical properties of program execution. We demonstrate the security improvement of statistical program behavior modeling through numerical analysis.

• We propose sFSA which models program behaviors by frequency distributions of n-step state transitions in FSA. To improve the sFSA’s scalability, we propose a novel method to project high dimensional n-gram system-call distributions onto a line and perform cluster analysis in the two-dimensional space.

• For more accurate and precise program behavior modeling, we propose sCFT, which models
the frequency distributions of n-step control-flow-transfers of a program execution. In particular, we employ the hardware-assisted tracing technique (i.e., Intel PT) to efficiently collect control-flow-transfers.

- We implement prototypes of our approaches and conduct an extensive experimental evaluation to demonstrate their effectiveness against real-world attacks and synthetic anomalies.

4.2 Motivation and Threat Model

4.2.1 Motivation

Given a program behavior model, identifying missed attacks potentially reveals the model’s weaknesses [148]. We define the attack freedom metric, which enables the comparison of detection capabilities among different models. Given a normal trace \( T \) (e.g., system-call trace) and a program anomaly behavior model \( M \), attack freedom refers to the number of permuted sequences (i.e., the attack trace) that \( M \) still accepts. The less attack freedom, the higher the security bar. In this section, we conduct a numerical analysis of attack freedom in different approaches, and demonstrate the security improvement of statistical program behavior modeling.

For demonstration purpose, we assume the program execution behaves in a deterministic manner at the system-call level, i.e., using a single system-call trace is sufficient to represent the normal behavior. For each run, we generate a random normal trace containing up to \( k \) types of system-call where the length of the trace is \( l \). We start with the n-gram model. Given the normal trace and the length of a n-gram (i.e., the value of \( n \)), we find evasive traces that could bypass the n-gram based anomaly detection. Among these abnormal traces that are evasive from the n-gram detection, we further find missed attack traces that can bypass both n-gram and distribution-based detection [42]. If the distribution of a testing trace has the same frequency distribution as the normal trace’s, we consider it a missed abnormal trace. For example, suppose the normal trace is \([1,3,4,4,5,2,4,3,4,3]\), where the length of the sequence is 10 and there are 5 different types of system-calls (indexed from 1 to 5). We find 31 evasive traces (i.e., the attack freedom is 31) for the 3-gram model, e.g., \([1,3,4,4,5,2,4,3,4,4]\). For the 1-step distribution model (i.e., [42, 40]), there are only 2 evasive traces out of the 31 attack traces in 3-gram model, i.e., \([1,3,4,3,4,4,5,2,4,3]\) and \([1,3,4,3,4,3,4,5,2]\). However, only one trace can be accepted by the 2-step distribution model, i.e., \([1,3,4,3,4,5,2,4,3]\).

Table 4.1 shows the attack freedom results of different program behavior models. For each setting, we run 1000 times repeatedly. The numbers in the table are the total evasive traces in 1000 runs. When the length of the trace \( l \) is 10, the types of system-calls \( k \) in the trace is 5, and the program behavior model is 2-gram, the 1-step distribution model [42, 40] significantly reduces the attack
<table>
<thead>
<tr>
<th>Setting (l,k,n)</th>
<th>N-gram</th>
<th>1-step distribution</th>
<th>2-step distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10, 5, 2)</td>
<td>2790855</td>
<td>107554 (3.8%)</td>
<td>17256 (0.6%)</td>
</tr>
<tr>
<td>(10, 5, 3)</td>
<td>19990</td>
<td>1284 (6.4%)</td>
<td>407 (2%)</td>
</tr>
<tr>
<td>(15, 5, 3)</td>
<td>569857</td>
<td>9966 (1.7%)</td>
<td>2561 (0.4%)</td>
</tr>
<tr>
<td>(15, 10, 3)</td>
<td>37769</td>
<td>255 (0.68%)</td>
<td>140 (0.37%)</td>
</tr>
<tr>
<td>(20, 10, 3)</td>
<td>426430</td>
<td>930 (0.21%)</td>
<td>509 (0.12%)</td>
</tr>
</tbody>
</table>

Table 4.1: Number of evasive cases (i.e., attack freedom) of different program behavior models

freedom by 96.2%. However, the attack freedom can be further reduced by 84% if using the 2-step distribution model. On average, the 2-step distribution model can reduce attack freedom by 65% over the 1-step distribution model. It is expected that even less room is left for mimicry attacks if we increase the step length. This motivates us to characterize the statistical properties of program execution for program anomaly detection with a finer detection granularity. The statistical anomaly detection improves the robustness against attacks by increasing the difficulties for a mimicry attack bypassing the anomaly detection.

Though the idea of statistical modeling is not completely new [41], this work distinguishes from existing frequency distribution-based methods in that: 1) We propose a statistical modeling framework that can be applied to any local models to detect frequency anomaly. 2) We present a novel method to project high dimensional frequency distributions onto a line for efficient cluster analysis; and 3) We systematically investigate the statistical program behavior modeling for anomaly detection. In particular, we show the effectiveness of using hardware-assisted tracing techniques (i.e., Intel PT) in our design.

4.2.2 Threat Model

With regards to the threat assumption, we have the same strong/conservative attack model as [148]. Attackers have partial or full knowledge about the deployed anomaly detector. To launch an evasive attack, the attacker needs to craft an exploit sequence (e.g., create a modified version of the malware by adding no-ops such as (void) getpid() or open(null, 0)) that is allowed by the detector without losing its functionality. For example, for the 1-step distribution model at the system-call level, attackers may provide a malicious software/patch to replace/modify the benign program. Such modified program can generate same system-call distributions as the benign traces, and thus evading from the detection. Our design objective is to improve the robustness of anomaly detection against data-oriented attacks by increasing the difficulties that an attack could bypass the anomaly detection.
4.3 Statistical Program Behavior Modeling

Fig. 4.1: Statistical program behavior modeling framework

Fig. 4.1 shows the statistical program behavior modeling framework. It is composed of two stages: training and detection, which are in line with many existing behavior-based anomaly detection systems. We propose two statistical program behavior models: 1) Statistical Finite-State Automaton (sFSA) model; and 2) Statistical Control-Flow-Transfer (sCFT) model. sFSA enhances the basic FSA model [87] by capturing frequency distributions of n-step state transitions in FSA. Frequency anomaly is the direct consequence of many non-control-data attacks in CPS programs, e.g., corrupting a local variable which controls the loop iterations of sending actuator signals. Thus, sFSA is particularly designed to efficiently detect frequency-based program anomalies in CPS. eFSA (in Chapter 3) and sFSA complement each other to detect a broad class of attacks that penetrate and modify the execution behaviors of CPS programs. sCFT models the frequency distributions of n-step control-flow-transfers of a program execution, which can be collected by utilizing existing program tracing infrastructures such as Intel’s Processor Trace (PT). It works at the control-flow-transfer level, and thus provides a finer granularity than sFSA. In sFSA and sCFT, we use cluster analysis to learn normal program behaviors from a set of frequency distributions. In the monitoring stage, an anomaly is marked if there is a significant difference between the testing trace and the normal program execution statistics (e.g., the distance is larger than a specified threshold).

4.3.1 Statistical Finite-State Automaton Model

FSA with Edge Weight

The basic FSA model (introduced in Section 3.4.1) only works if an attack causes the program execution to deviate from the learnt automaton. Stealthy exploits (e.g., control-flow bending [3]
and non-control-data attack [21]) that achieve attack goals without violating the program’s control-flow integrity can evade from the detection. Statistical modeling of program behaviors has the potential to detect such attacks, by differentiating the frequencies of occurrences among feasible control-flow paths.

We first obtain state transition frequencies across normal program traces, where edges in FSA are associated with probability values, similar to the construction of a probabilistic finite-state automaton. Fig. 4.2(a) illustrates an example of the basic FSA model augmented with state transition frequencies. As shown in the figure, branches are not uniformly distributed, but occur with different frequencies. Because individual paths are normally executed with different frequencies in practical program runs.

![Figure 4.2: (a) FSA augmented with state transition frequencies; (b) Calculation of n-step state transition probabilities](image)

**sFSA – Learning Distribution of State Transitions**

We characterize the statistical properties of a behavior instance in terms of the distributions of *n-step state transitions*, where *n* is called as step length. In Section 3.4.4, the execution window is defined as one top-level loop iteration in a continuous program in CPS. A behavior instance is termed as the program activity within an execution window. For other programs, we define the execution window as a semantically meaningful portion of a program execution, e.g., by functions, address ranges, or the whole program. The idea is to model normal executions of a program by frequency distributions of finite length state transitions in FSA, where each n-step transition corresponds to a high-level execution feature (*i.e.*, one dimension) in our behavior model. Then, cluster analysis can be used to learn distinct normal program behaviors from a set of distributions [42, 40]. However, such distribution-based profiling is challenging since the theoretical maximum number of different
n-step transitions is $k^n$, where $k$ is the number of states in FSA. When $k$ is large, the dimensions become quite large, which incurs a very high analysis overhead.

To reduce computational complexity, we project high dimensional distributions onto a cdf (cumulative distribution function) curve in the two-dimensional space, which captures the occurrence frequency of specific n-step state transitions. For each behavior instance, we first calculate all n-step state transition probabilities based on a sliding window with step length $n$. Take the example in Fig. 4.2(b), for a behavior instance $3 \rightarrow 6 \rightarrow 7 \rightarrow 6 \rightarrow 7 \rightarrow 10 \rightarrow 11$, we calculate 3-step state transition probabilities by multiplying corresponding edge weights associated with the state transitions. In this example, we obtain four probability values (since we find four 3-step transitions). Then, we convert these probability values to a cdf curve with values in the range $[0, 1]$. In sFSA, we use fixed number of sample points to describe a cdf line. As a result, we transform the cluster analysis of high/unbounded-dimensional distribution data into clustering of a set of cdf curves with a limited number of sample points.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{c.png}
\caption{Behavior instances are transformed into cdf curves and then clustering is performed.}
\end{figure}

Fig. 4.3 presents an example of the sFSA behavior model. Given the normal behavior instances in Fig. 4.3(a), Fig. 4.3(b) plots the output clusters (represented by cdf curves) after performing x-means clustering [149]. In this example, cdf curves are derived by calculating probabilities of 3-step state transitions. The center of each of x-clusters corresponds to a normal program execution context. Now, given a new behavior instance observed in the testing phase, we convert it into a cdf curve and measure the average distance between the new observation and each of sFSA’s cluster centers, which is calculated as the average value of all Euclidean distances at cdf sample points. We define the cluster classification threshold where if the average distance between a new cdf curve and a cluster center in sFSA is smaller than the threshold, we consider this cdf curve belongs to the cluster. Fig. 4.3(b) illustrates an example of frequency anomaly, where the number of iterations
of loop $S_2 S_3 S_3$ is abnormally large. When we set the cluster classification threshold to 0.01, the testing curve does not belong to any existing cluster. Thus, it is considered an anomaly by our $s$FSA detection.

4.3.2 Statistical Control-Flow-Transfer Model

Motivation

Our $\varepsilon$FSA and $s$FSA work at the granularity of system-calls. However, if an attack does not involve changes of system-call usage, then it can completely bypass the detection. Code 4.1 shows an example where the data-oriented attack can evade the detection from system-call level monitoring. In Code 4.1, a control program sends control signals by directly writing registers without calling any system-call using the WiringPi library on Raspberry Pi. Tracing control-flow-transfers of a program provides a more accurate characterization of program behaviors. Existing tracing facilities make it possible to efficiently monitor control-flow-transfers at runtime, such as using Intel’s Processor Trace (PT) and ARM’s CoreSight techniques. In this section, we employ PT for tracing control-flow-transfers and demonstrate the detection capability of $s$CFT model against data-oriented attacks.

```
#include <wiringPi.h>
int main (void) {
    ...
    if ( condition1 ) {
        digitalWrite (0, HIGH) ; //no system call
        delay (500) ;
        digitalWrite (0, LOW) ;
        delay (500) ;
    } else if ( condition2 ) {
        digitalWrite (1, LOW) ;
        delay (500) ;
        digitalWrite (1, HIGH) ;
        delay (500) ;
    }
}
```

Code 4.1: A control branch attack evades from the system-call level monitoring
bool do_authentication(char* str) {
    char password[10] = "";
    int authenticated = 0;
    strcpy(password, str);
    if (auth_password(password)) {
        authenticated = 1;
    }
    if (authenticated)
        return true;
    else
        return false;
}

bool auth_password(char* password) {
    if (!strcmp(password, "password"))
        return true;
    else return false;
}

int main(int argc, char *argv[]) {
    if (do_authentication(argv[1])){
        printf("Authentication successful\n");
        return 1;
    } else {
        printf("Authentication failed\n");
        return 0;
    }
}

Figure 4.4: An example that simplifies the attack on the authentication flag in SSH server [1]

Tracing Control-Flow Transfers Using PT

Fig. 4.4 shows an example of the data-oriented attack, which simplifies the attack on the authentication flag in SSH server [1]. A stack flag variable authenticated is used to indicate whether a remote user has passed the authentication. When the vulnerable function strcpy is called (in line 4), an attacker is able to corrupt the authenticated flag to a nonzero value, which leads to an always successful user authentication (in line 8).

Figure 4.5: An example of program execution traces (i.e., user space PT events) collected by PT corresponding to the code in Fig. 4.4.
Fig. 4.5 shows an example of program execution traces collected by PT (only the user space control-flow transfers are shown), corresponding to the example code in Fig. 4.4. As illustrated in the figure, the nodes include both indirect (e.g., function-call) and conditional branches, and the edge captures the transition between two nodes. Note that the TNT packet is generated based on the assembly code, rather than the source code. An example is given in Fig. 4.6. When the true-branch of if(!strcmp(password,"password")) (in line 14 of Fig. 4.4) is executed, PT captures a non-taken (i.e., N) event. Scrutiny of the assembly code reveals the reason. The conditional branch corresponds to a conditional jump jne in assembly code. If jne is non-taken, the branch in line 14 returns true (i.e., which means the true-branch is taken in source code).

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048484:</td>
<td>85 c0</td>
<td>test %eax,%eax</td>
</tr>
<tr>
<td>8048486:</td>
<td>75 07 jne</td>
<td>804848f &lt;auth_password+0x24&gt;</td>
</tr>
<tr>
<td>8048488:</td>
<td>b8 01 00 00 00</td>
<td>mov $0x1,%eax</td>
</tr>
<tr>
<td>804848d:</td>
<td>eb 05 jmp</td>
<td>8048494 &lt;auth_password+0x29&gt;</td>
</tr>
<tr>
<td>804848f:</td>
<td>b8 00 00 00 00</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>8048494:</td>
<td>c9 leave</td>
<td></td>
</tr>
<tr>
<td>8048495:</td>
<td>c3 ret</td>
<td>Assembly Code</td>
</tr>
</tbody>
</table>

Return False
8048486 => 804848f //Taken (T)
8048495 => 80484d5 //TIP 0x80484d5

Return True
804848d => 8048494 //Not Taken (N)
8048495 => 80484d5 //TIP 0x80484d5

Figure 4.6: Examples of TNT packets and the corresponding assembly code

sCFT – Learning Distribution of Control-Flow Transfers

We decode PT’s raw TIP/TNT events and reconstruct of control-flow-transfers of the program execution in each execution window. Similar to the sFSA model, sCFT learns the statistical properties of control-flow-transfers of each behavior instance. For the example code illustrated in Fig. 4.5, we consider all control-transfers of the whole program execution as a behavior instance.

In the training phase, we conduct a cluster analysis to learn distinct normal behaviors. To reduce analysis overhead, we apply the Principal Component Analysis (PCA) technique for dimension reduction. We then adopt the x-means clustering approach [149] to cluster all behavior instances, where the center of each of the x-clusters represents a normal program execution context. In the monitoring phase, we detect abnormal executions if the statistics in the testing trace vary substantially from the normal clusters.

4.4 Experimental Validation

We evaluate sFSA and sCFT’s detection capabilities against data-oriented attacks from multiple perspectives. Our evaluation aims to answer the following questions:

- Can sFSA detect frequency anomalies caused by data-oriented CPS attack?
What is the detection accuracy of sFSA when using crafted synthetic program traces?

What is the detection capability of sCFT in comparison to sFSA?

What is the runtime performance overhead of sFSA and sCFT?

4.4.1 Experiment Setup

We first evaluate the detection capability of sFSA against the control-intensity attack in CPS. The experiment setting of the CPS program is same as in Section 3.7. Then, we study the performance of sFSA using a utility program grep with 809 test cases from the Software-artifact Infrastructure Repository (SIR). We choose grep for testing because it comes with comprehensive test cases in SIR. Finally, we conduct a comparison between sCFT and sFSA in terms of detection capability and performance overhead.

To construct statistical models, we adopt the x-means clustering algorithm [149] to cluster all behavior instances in normal program traces in the training phase. X-means clustering automatically determines the number of clusters based on Bayesian Information Criterion (BIC) scores. We used the PyClustering library for x-means clustering in sFSA.

4.4.2 Detecting Data-Oriented Attack in CPS

In this experiment, we evaluate sFSA against the CPS control-intensity attack using the SyringePump application. We first perform a loop detection to identify top-level loop boundaries in traces, and partition these infinite traces into a set of behavior instances. In building the sFSA model, we set the number of sample points for describing a cdf line to 30. We adopt the default values of system parameters (e.g., tolerance, criterion, ccore) for x-means clustering according to the PyClustering library. The step length of calculating the n-step transition probability in sFSA is set 3. In the testing phase, to decide whether a new cdf curve belongs to a cluster in sFSA, we set the classification threshold to 0.01. That is, we consider the cdf curve belongs to a cluster if their average distance is less than 0.01.

In the training phase, we collect execution traces of SyringePump running in an indoor laboratory environment. From the training dataset, the normal values of relative humidity in our experimental environment ranged between 20rH and 38rH. We set the threshold that triggers the movement of syringe pump to be 30rH, where the amount of liquid to be dispensed equals to the instantaneous

---

1http://sir.unl.edu/content/bios/grep.php
2http://pythonhosted.org/pyclustering/
humidity value subtracted by $30 rH$ in this test. We manipulate the humidity values to be in the range from $50 rH$ to $55 rH$. As a result, the SyringePump sends more control signals to dispense more liquid than expected, which incurs frequency anomalies.

An example of the testing phase in SyringePump experiment is shown in Fig. 4.7. From the training dataset, the normal values of relative humidity in our experimental environment ranged between $20 rH$ and $38 rH$. When the humidity value is manipulated to a larger value, the number of loop iterations to send syringe pump moving signals is also increased, which is reflected by the number of system-calls in each behavior instance. Fig. 4.7 plots the cdf curves in the $sFSA$ model corresponding to a series of behavior instances. We observe two distinctive groups of cdf curves. $sFSA$ is able to detect the frequency anomalies in our experiments.

### 4.4.3 Systematic Accuracy Evaluation

To thoroughly evaluate how sensitive and accurate the $sFSA$ model is, we investigate the tradeoff between detection rates and false positives, by changing i) the step length of calculating n-step state transition probability from 3 to 7; ii) the number of sample points for describing a cdf line from 10 to 50; and iii) the cluster classification threshold from 0.01 to 0.15. We compare $sFSA$ with the n-gram detection approach, where normal patterns are based on n-step system-call sequences, and $n$ is varied from 3 to 7.

To measure false positives, we perform 10-fold cross-validation on the normal traces. To evaluate the detection rates against stealthy anomalies without violating the program’s control-flow integrity,
we generate \textit{synthetic traces}, which simulate a wide variety of potentially stealthy attacks. In particular, we develop a tool to find \textit{non-control sub-loops} in normal traces, which enables us to simulate data-oriented attacks (\textit{i.e.}, generating frequency anomalies) without prior knowledge of the vulnerabilities. In this work, a non-control sub-loop is defined as a subset of traces (\textit{i.e.}, a subsequence of state transitions), after either being deleted from or inserted into the original normal trace, the new synthetic trace does not violate FSA’s state transition integrity. Take Fig. 4.2 for example, the inner loop $S_6 S_7 \ldots S_6 S_7 S_6 S_7$ is a typical non-control sub-loop. Inserting or deleting such gadgets will generate synthetic traces with frequency anomalies. In this experiment, we sort all non-control sub-loops identified by our tool in a descending order in terms of the sub-loop length, and select the first 100 sub-loops for insertion or deletion to generate the synthetic traces.

Fig. 4.8 plots the tradeoff between detection rates and false positives when applying sFSA on \texttt{grep}. Each point in the figure is associated with a specific configuration. The results have been averaged based on multi-round 10-fold cross-validation with 809 test cases. The figure clearly demonstrates the effectiveness of our approach in detecting various stealthy anomalies, and it achieves much higher detection rates than the n-gram approach. Although n-gram introduces very small false positives, it yields a poor detection rate (less than 50% in all cases) since inserting or deleting non-control sub-loops may not result in abnormal system-call subsequences/n-grams.

In Fig. 4.9, we study the impact of step length on sFSA’s false positives. We fix the number of sample points to 30, and the cluster classification threshold to 0.01. As shown in Fig. 4.9, when the step length of calculating the n-step transition probability is set 3, its false positive was higher than 14%. With increasing the step length, the false positive rate decreases significantly. This is
because, in `grep`, using 3-step transition probabilities does not generate distinctive cdf curves in learning distributions of state transition in sFSA. Fig. 4.10 shows the training delay of sFSA. It takes on average less than 1s to complete the training of more than 700 `grep` traces (i.e., each round of the 10-fold cross-validation).

### 4.4.4 Comparison of sCFT and sFSA

In this experiment, we use sCFT and sFSA to detect the non-control data attack that manipulates the authenticated flag to bypass the user authentication process in SSH server. Instead of using the original code, we craft a self-contained model program that preserves the semantics of the authentication process, as shown in Fig. 4.4. One reason is that the crafted small program does not trigger distinctive system-calls along different branches, so that we demonstrate the benefit of branch-level control-flow tracing offered by PT compared with the system-call tracing. There are two normal execution paths, successful authentication and failed authentication, respectively. We collect user space control-flow traces using the Linux `perf` on a commodity Intel Skylake machine with Intel PT support.

The program involves 55 distinctive 2-step control-flow-transfers in user space excluding library addresses, after applying PCA, we reduce the PT trace data to 3-dimensional data and then perform the x-clustering. In this example, the training dataset has been clustered into 2 clusters (i.e., the blue dots in Fig. 4.11). The attack trace shows a different TNT event sequence from the normal traces. As shown in Fig. 4.11, the distance between the attack instance (i.e., the red triangle) to the normal clusters is 3.7 and 5.2, respectively, where we can easily detect the anomaly. However, for the sFSA model, the attack instance has an exactly same trace (i.e., totally 26 system-calls) as the normal successful authentication instance at the system-call level. Thus, sFSA fails to detect the anomaly.
Then, we compare the runtime performance overhead of Linux perf (version 4.13 with PT support) and Strace (version 4.13 with stack unwinding support enabled). The program and test cases used in the experiment include the utility program grep and its 809 test cases from the Software-artifact Infrastructure Repository (SIR) benchmark suite [95]. We repeatedly run the 809 test cases for 100 times, and collect the program execution time with and without using tracing tools, respectively. The testing platform is a desktop computer with an Intel i7-6700K Skylake CPU @ 4.00GHz, running 32bit Ubuntu OS. All the results have been averaged over 809×100 runs, and the related standard deviations are provided as error bars. The baseline refers to the execution time without running any tracing tool. It shows that Strace tracing incurs 127% slowdown, while the hardware-assisted tracing (i.e., perf for PT) only incurs 4.5% tracing overhead. The size of raw trace data ranges from 0.03MByte to 0.1MByte for PT tracing.

Table 4.2: Summary of sFSA and sCFT models

<table>
<thead>
<tr>
<th>Model</th>
<th>Tracing granularity</th>
<th>Infrastructure requirement</th>
<th>Slowdown overhead</th>
<th>Storage overhead</th>
<th>Detection Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>sFSA (w/ strace)</td>
<td>◙</td>
<td>◙</td>
<td>◙</td>
<td>◙</td>
<td>◙</td>
</tr>
<tr>
<td>sCFT (w/ PT)</td>
<td>◙</td>
<td>◙</td>
<td>◙</td>
<td>◙</td>
<td>◙</td>
</tr>
</tbody>
</table>

*: Low  ◙: Medium  ◙: High

Table 4.2 shows a summary of the comparison between sFSA and sCFT. sCFT provides a finer granularity and better detection capability than sFSA. But it requires the hardware tracing infrastructure...
support (e.g., modern Intel processors). Though control-flow tracing (with hardware-assisted tracing support) shows a low runtime performance overhead, it incurs a large storage overhead.

4.5 Summary

In this work, we presented a general framework for fine-grained program anomaly detection by characterizing statistical properties of program execution at different granularities. We used the attack freedom metric to measure the theoretical detection capability of different detection models. Through numerical analysis, we demonstrated the security improvement of the statistical program behavior modeling. Then, we proposed two statistical program behavior models: $s$FSA and $s$CFT. $s$FSA models program behaviors by frequency distributions of n-step state transitions in FSA at the system-call level. $s$FSA is an ideal model that complements $e$FSA (in Chapter 3) to detect data-oriented attacks in CPS. To improve $s$FSA’s scalability, we presented a novel method to project high dimensional n-gram system-call distributions onto a line and performed cluster analysis in the two-dimensional space. $s$CFT models the frequency distributions of n-step control-flow-transfers of a program execution. It further raises the security bar for more accurate and precise program behavior modeling. An extensive experimental evaluation demonstrated that statistical anomaly detection improves the robustness against data-oriented attacks by increasing the barriers that an attack could bypass the anomaly detection.
Chapter 5

Defending Against Data-Oriented Programming (DOP) Attacks

5.1 Introduction

Data-oriented programming (DOP) is a recently proposed technique to construct expressive (i.e., towards Turing-complete computation) data-oriented attacks for arbitrary x86 programs [21]. The key idea of DOP exploit is to misinterpret existing short code sequences (i.e., data-oriented gadgets) for malicious purposes. It allows a vulnerable program to perform arbitrary computations in program memory under the control of a remote adversary.

Same as other memory-corruption attacks such as control-flow attacks, DOP relies on the presence of memory corruption vulnerabilities. Despite considerable prior research in retrofitting memory-unsafe programs with memory safety guarantees, memory corruption problems would persist for a long time, since memory-unsafe languages such as C/C++ are still widely used today for the reason of better interoperability and speed performance. On the other hand, though data-oriented attacks are known for a long time, the threats remain insufficiently addressed. As launching a control-flow attack becomes more and more difficult due to many deployed defenses against control-flow exploits, data-oriented attacks have emerged to be an attractive attack technique for cyber attacks.

Program anomaly detection is considered as an effective defense mechanism against software attacks, which complements existing mitigation techniques, e.g., memory safety enforcement. However, traditional software-based program tracing has been a serious performance bottleneck for anomaly detection, which hinders the practical deployment of program anomaly detection systems. To overcome this limitation, researchers proposed to adopt a hardware-based approach to monitor program’s executions [96]. PT is a new hardware feature provided by commodity Intel Skylake.
machines to trace program control-flows, with less than 5% runtime overhead. This low-overhead tracing technique has received considerable attention because it has the potential to bring program anomaly detection to the practical deployment.

In this work, we focus on program anomaly detection against DOP attacks using PT tracing. The main technical contributions are summarized as follows.

- We demystify the DOP exploitation technique and show its complexity and rich expressiveness, through a case study of a real-world ProFTPd DOP attack. Then, we discuss the detectability of PT-based program anomaly detection against data-oriented attacks.

- We propose a \textit{D\textsubscript{E}DOP} anomaly detection system, which enforces the branch correlation integrity and improves anomaly detection sensitivity by capturing statistical characteristics of both short control-transfers and client-server interactions.

- To derive branch correlations in a program, we develop a general branch correlation analysis tool based on LLVM\cite{llvm} to automatically identify coarse-grained correlated branches. Then, we use the satisfiability modulo theories (SMT) logic solver to identify strong correlated branches.

- Through experimental validation, we show that though DOP can be used to construct expressive (Turing-complete) non-control data attacks, they are less evasive from detection than the basic non-control-data attacks. Results demonstrate that \textit{D\textsubscript{E}DOP} can successfully detect program behavior anomalies in different dimensions against the real-world ProFTPd DOP attack, and it can also be applied to detect basic non-control-data attacks.

5.2 ProFTPd DOP Attack Demystified

To study the problem of defending against DOP attacks, we first need to understand the process of launching such an exploit. We use the ProFTPd DOP attack crafted by Hu et al.\cite{hu2016} to illustrate the typical attack flow of DOP attacks. The goal of this DOP attack is to bypass randomization defenses (such as ASLR), and then leak the server’s OpenSSL private key. The private key is stored on the heap and accessed via a chain of 8 pointers. Though the key is stored in a randomized memory region due to ASLR, the base pointer is not randomized because the Position Independent Executable (PIE) is disabled by default on 32bit OS\cite{pie}. Therefore, it makes possible to exfiltrate the private key by starting from the OpenSSL context base pointer (\textit{i.e.,} a known location of the static variable \texttt{ssl\_ctx}) and recursively de-referencing 7 times within the server’s memory space.
ProFTPd Vulnerability

ProFTPD versions 1.2 and 1.3 have a stack-based buffer overflow vulnerability in the `sreplace` function (CVE-2006-5815 [150]). The overflow can be exploited by an attacker to obtain an arbitrary write primitive. The server program provides a feature to display customized messages when a user enters a directory. The content of messages is saved in `.message` file in each directory. It can be edited by any user having write-access to the directory. The `.message` file can contain special characters which will be replaced with dynamic content such as time/date and server name by the `sreplace` function. For example, the string "%V" in `.message` will be replaced by `main_server->ServerName`, and "%T" will be replaced by the current time and date. Changing the working directory with a `CWD` command triggers the processing of `.message` file, and subsequently triggers the invocation of `sreplace` function. Thus, the attacker crafts attack payloads by modifying the content of `.message` file on the server, and then sending `CWD` commands to trigger the vulnerable function.

```c
char *ssstrncpy(char *dest, const char *src, size_t n) {
    register char *d = dest;
    for (; *src && n > 1; n--)  
        *d++ = *src++;
    ...
}
char *sreplace(char *s, ...) {
    ...
    char *m, *r, *src = s, *cp;
    char **mptr, **rptr;
    char *marr[33], *rarr[33];
    char buf[BUF_MAX] = {'\0'}, *pbuf = NULL;
    size_t mlen=0, rlen=0, blen; cp=buf;
    ...
    while(*src){
        for(mptr=marr, rptr=rarr; *mptr; mptr++, rptr++) {
            mlen = strlen(*mptr);
            rlen = strlen(*rptr);
            if(strncmp(src,*mptr,mlen) == 0){
                ssstrncpy(cp,*rptr,blen-strlen(pbuf));
                if(((cp + rlen) - pbuf + 1) > blen){
                    cp = pbuf + blen - 1;
                    /*Overflow Check*/
                }
            ...
            src += mlen;
            break;
            }
        }
    }
}```
if(!*mptr) {
    if((cp - pbuf + 1) > blen){
        cp = pbuf + blen - 1;
        /*Overflow Check*/
    }
    *cp++ = *src++;

} }

Code 5.1: The vulnerable function in ProFTPd server

Code 5.1 shows the vulnerable `sreplace` function. The vulnerability is introduced by an off-by-one comparison bug in line 30, which allows attackers to modify the program memory. A defective overflow check in lines 29-34 is performed to detect any attempt to write outside of the buffer boundary. When writing to the last character of the buffer `buf`, `(cp-pbuf+1)` equals to `blen`. Thus, the predicate in line 30 returns `false`, and the write to `buf`'s last character in line 33 is followed. Subsequently, the string is not properly terminated inside the buffer because the buffer's last character has been overwritten with a non-zero byte. In the next iteration of the `while` loop, the input of `strncpy` in line 20 `blen-strlen(pbuf)` becomes negative, which will be interpreted as a large unsigned integer by `strncpy` function. Hence, the invocation of `strncpy` overflows outside buffer bounds into the stack and overwrites local variables such as the content of `rarr` and `cp`. As a result, both the source (i.e., `*rptr`) and the target (i.e., `cp`) of the string copy `strncpy` in line 20 are corrupted, which allows the attacker to control the source, destination, and the number of bytes copied by subsequent iterations of the `while` loop in lines 15-35.

The Attack Flow

Fig. 5.1 shows a step-by-step description of the ProFTPd DOP attack that bypasses randomization defenses. The attacker interacts with the server program to corrupt program memory repeatedly (over the course of numerous FTP commands) exploiting the buffer overflow vulnerability. In this scenario, the command handler `cmd_loop` in ProFTPd serves as the data-oriented gadget dispatcher. In each iteration, the attacker triggers the execution of targeted gadgets by sending a crafted attack payload to the server program. The dereference gadget `d++=*src++` (line 4 in Code 5.1) is located in function `strncpy`. We reproduced the ProFTPd DOP attack, and observed that the vulnerable function `sreplace` is called more than 180 times during the attack. Over the course of the attack, the attacker systematically corrupts program memory to construct a DOP program out of individual operations that perform the following steps.

1. To read data from arbitrary addresses in the server, an attacker needs to overwrite string
pointers used by a public output function (e.g., send). To this end, the attacker manipulates 12 pointers in a local static mons array located at 0x80cf6e0 to a global writable location (i.e., the attacker specifies this location, denoted by G_PTR). As shown in Fig. 5.1, the mons array is filled with G_PTR’s address 0x80d3450. When the server returns month information to the client, it actually prints the value pointed by G_PTR. This step builds an exfiltration channel which can leak information from the server to the network.

The attacker knows the memory address of the global pointer main_server at 0x80d6e14, which points to the main server structure. Then, the attacker reads the structure address pointed by main_server, i.e., 0x871ae3c. The read operation is implemented by writing the address of the main server structure to the global writable location G_PTR, and then transmitting the output via the exfiltration channel to the attacker side.

The attacker knows the offset of the field ServerName to the base address of the main server structure, which is 0x10 according to the binary of ProFTPD. Then, the attacker is able to calculate the address of main_server->ServerName, i.e., 0x871ae3c+0x10=0x871ae4c. Given the memory address 0x80de0c8 of ssl_ctx, i.e., the base pointer of a chain of 8 pointers to the private key, the attacker writes this address to main_server->ServerName located at 0x871ae4c.

D Dereference the base pointer ssl_ctx, where the output is 0x874d7b8. The dereferencing operation dereferences the value currently located at main_server->ServerName, by triggering the execution of the dereference gadget in line 4 of Code 5.1. The dereferenced value will be copied to a known position in the response buffer resp_buf. Then, the attacker obtains the address 0x874d868 of cert by adding the offset 0xb0 to the dereferenced value.

Figure 5.1: ProFTPd DOP attack flow
0x874d7b8. After that, the attacker copies the address of `cert` to `main_server->ServerName` for the next iteration of dereference. This step repeats 7 times following the dereference chain shown in Fig. 5.1. The offset of the relevant field to the base address in each iteration can be derived from the binary or source code. Finally, the final address of the private key is obtained.

6. Sequentially read 8 bytes from the private key buffer via the covert channel constructed in the first step. This process repeats for 64 times to retrieve totally 512 bytes data.

**Threat Model**

A DOP exploit typically involves multiple-step data manipulations. It requires non-trivial engineering efforts to construct DOP gadgets and chain them for malicious effect. We assume the presence of a memory corruption vulnerability (such as a buffer or heap overflow) in the target program, which enables attackers to modify the content of the application’s memory. Attackers have an in-depth knowledge of the vulnerable program’s exact memory layout at runtime. They have identified data-oriented gadgets and gadget dispatchers, and can stitch these gadgets and then set up the execution of gadgets in sequence. We are not concerned about how attackers successfully gained entry into the system and carried out a DOP attack, but focus on detecting abnormal program behaviors given the presence of the attack.

### 5.3 Detection of DOP Attacks

#### 5.3.1 Observations

![Figure 5.2: Three types of anomalous control-flow behaviors caused in DOP attacks](image)

The *necessary condition* of detecting a DOP attack using control-flow tracing is that, the attack directly or indirectly affects the flow of a program’s execution. We observe that DOP attacks can
potentially cause three types of anomalous control-flow behaviors, as shown in Fig. 5.2. The first two types also apply to DDM attacks.

* **Incompatible branch behavior:** Manipulating a predicate-use variable (e.g., decision-making data and user identity data) can change the default branch behavior of a program. If there exist two correlated conditional branches that are data-dependent on the manipulated variable before and after the data manipulation site, and there is no write to the manipulated variable between these two branches, it is likely the data manipulation incurs incompatible branch behavior that can be detected by control-flow tracing. Such anomalies exhibit in Stage S3 of data-oriented attacks (i.e., when a corrupted variable is used). Continuous buffer overflow may generate side impacts on control-flow behavior, which may result in an incompatible branch behavior observable in Stage S2. For example, though the target buffer is not used for predicate-use, some decision-making variables close to the buffer may be inevitably corrupted. We manually analyzed 14 vulnerable programs in a test Suite for buffer overflows [151], and found that 5 out of 14 overflows cause side impacts on decision-make variables (i.e., involved in predicate expressions).

* **Micro-level control-flow frequency anomaly:** It refers to unusual execution frequencies of short control-flow paths. For instance, corrupting variables which directly or indirectly control loop iterations cause such frequency anomalies. Micro-level control-flow frequency anomalies may be observed in different stages of data-oriented attacks. In addition, Control-Flow Bending (CFB) attacks [3] and resource wastage attacks [16] may also lead to unusual control-flow frequencies. Such control-flow frequency anomalies can happen in all stages in the course of a data-oriented attack. For example, if a continuous buffer overflow corrupts a variable that controls the amount of a loop iteration, an abnormal pattern of the loop usage may be observed in Stage S2. In Section 5.5.1, we show triggering a memory error in Stage S1 may also incur the micro-level control-flow frequency anomaly.

* **Interaction frequency anomaly (macro-level frequency anomaly):** In an interactive DOP attack, an attacker interacts with a vulnerable program to repeatedly corrupt variables to achieve the attack purpose and avoid segmentation faults. This inevitably results in frequency anomalies during the client-server interaction, which can be captured by control-flow tracing. For example, in the ProFTPd DOP attack introduced in Section 5.2, an attacker needs to send a large number of FTP commands with malicious inputs to the ProFTPd server to corrupt the program memory repeatedly.
5.3.2 DEDOP Anomaly Detection System

In this section, we present the DEDOP anomaly detection system to defend against DOP attacks. Fig. 5.3 shows the design overview of DE DOP, which is composed of two stages: offline training and online monitoring. In the training phase, DE DOP enforces the branch correlation integrity to detect any incompatible branch behavior, and detects frequency anomalies of both short control-transfers and client-server interactions.

Branch correlation integrity validates whether a conditional branch execution (branch taken or non-taken) is consistent with the outcomes of other correlated conditional branches. In particular, we use program dependency analysis and the satisfiability modulo theories (SMT) logic solver to identify correlated branches, which serve as the inherent checkpoints to detect any violation of the branch correlation integrity. At runtime, DE DOP maintains a branch status table to keep track of correlated branch behaviors [100]. For example, given a correlated branch pair <BR1, BR2> and their branch correlation rule, when DE DOP observes that BR1 is taken, it derives the expected outcome of BR2 and updates the branch status table accordingly. Later if the branch outcome of BR2 is different from the expected one, DE DOP raises an alarm.

sCFT model (introduced in Chapter 4) characterizes the statistical properties of control-transfers, which can be applied to detect the micro-level control-flow frequency anomaly. We employ the similar modeling technique to model the client-server interaction behaviors by frequency distributions of short FTP command sequences (i.e., n-grams of FTP commands). In the monitoring phase, we first detect any incompatible branch behavior, and then detect any control-transfer frequency anomaly.
5.4 Branch Correlation Analysis

In this section, we present a general method to identify coarse-grained correlated conditional branches that have either direct or indirect joint data-dependency. Then, we use the satisfiability modulo theories (SMT) logic solver to find "strong" correlated branches. It is worthy mentioning that the correlated branches identified after applying the SMT logic solver achieves a zero false positive rate.

5.4.1 Challenges

Conditional branches are optional execution paths. Prior studies show that conditional branches are not completely independent of each other [100]. In many instances, a branch’s outcome can be correlated by other branches’ outcomes. Specially, we use the "strong correlation" to indicate that a branch’s outcome can be derived from other branches’ outcomes. Branch correlations have been well investigated for compiler optimizations (such as static branch prediction and elimination) [152] [153]. Zhang et al. [100] exploited correlations among branches to detect infeasible program paths caused by attacks. However, their approach is restricted to simple forms of intra-procedural conditional predicates, e.g., comparing a variable with a constant value. As a result, only limited branch correlations can be explored for anomaly detection in [100].

It is challenging to statically determine complicated branch correlations with arbitrary predicate expressions, especially for predicates with indirect data-dependency or involving function calls. We identify the following scenarios that make branch correlation analysis non-trivial.

- Complex predicate expression: A branch predicate may be the combination of multiple conditions, and involves multiple variables and operands. It is difficult to directly derive the "subsume" or "mutually exclusive" relationship between branches with complex expressions. If the predicate coverage of one branch BR1 is no less than the predicate coverage of another branch BR2, we say BR1 "subsumes" BR2. BR1 and BR2 are "mutually exclusive" if there is no overlap between their predicate coverages.

- Indirect correlation: Two branches may be correlated with each other without having a direct connection (e.g., using the same predicate variable). For example, two branches use different predicate variables, but these variables exhibit a data-dependency. Another scenario is that the outcome of one branch changes the condition determining the outcome of another branch.

- Inter-procedural correlation: Branches in different functions can be correlated if there exists an inter-procedural data-dependency between predicate variables. Such a correlation usually has a longer distance.
### 5.4.2 Identifying Correlated Branches with Static Analysis

In order to capture branch correlations with arbitrary predicate expressions, we consider the data and control-flow properties of conditional branches. Our LLVM-based [47] algorithm automatically identifies coarse-grained correlated branches. We first obtain the LLVM Intermediate Representation (IR) of a program, and construct the program dependence graph, including both data and control dependencies. Our analysis method is compatible with multiple programming languages since LLVM supports a wide range of languages. For clarity purpose, in the following, we elaborate how our algorithm handles complex predicates with specific examples using C-based programs.

![Figure 5.4: Examples of direct and indirect branch correlations](image)

Fig. 5.4(a) shows an example of direct correlations between two branches with simple forms of predicates. We can easily identify such cases through backward data-dependency analysis, as long as they share the same variable declaration instruction. To capture indirect correlations, for each conditional branch instruction (i.e., the conditional "br" instruction with two "label" values), we trace back along with its backward data-dependencies to find the original declarations of variables involved in the predicate expression. For example in Fig. 5.4(b), starting from branch BR1, along with its backward data-dependence slice, we find variable \( b \)'s use (load) instruction (1). We conduct the forward data-dependence search to find any "store" instruction, which indicates a data assignment to \( b \) (i.e., the right operand). As shown in Fig. 5.4(b), we find a "store" instruction regarding the statement \( b=a \) (2). From there, we then check the "store" instruction’s backward data-dependence slice to find the declaration instruction of \( a \) (3). After that, we conduct the forward data-dependence search to find any conditional branch instruction related to \( b \) (4). Finally, we reach the branch instruction regarding branch BR2, and find a coarse-grained correlated branch pair <BR1, BR2>.

To identify inter-procedure branch correlations, for each conditional branch instruction, we first obtain its backward data-dependent "load" instructions (1). Then, we search any function call
Figure 5.5: Example of inter-procedure branch correlation

instruction along with the backward control-dependent path from the "load" instruction (2). As shown in Fig. 5.5, we find that the "load" instruction regarding \( flag \) is control-dependent on \( func \). Similar to the procedure of identifying indirect correlations, through backward data-dependence search, we find the declaration instruction of \( a \) (3). Then, the forward data-dependence search finds the branch instruction regarding branch BR2 (4). We find that BR2 is an inter-procedure correlated branch with BR1.

5.4.3 Identifying Strong Branch Correlation

The objective of our branch correlation analysis is to find out whether a branch’s outcome can be derived from other correlated branches’ outcomes. Using correlation information, we are able to detect incompatible branch behaviors at runtime. However, the above analysis only identifies coarse-grained correlated branches that have joint data-dependency. In order to avoid false positives, we define the strong branch correlation as the case if the outcome of a branch can be always predicted by another coarse-grained correlated branch’s outcome. Strongly correlated branches should exhibit fixed runtime control-flow patterns. Note that branch correlation has the one-way property, \( i.e., \) when BR1’s outcome can be derived from the outcome of BR2, the opposite may not be true.

In this work, we use the satisfiability modulo theories (SMT) logic solver, \( e.g., \) Z3 [154], to determine any "subsume" or "mutually exclusive" relationship between branches. As for our future work, we plan to utilize sequential pattern mining techniques based on dynamic profiling to obtain the one-way \( true/false \rightarrow true/false \) patterns for complex correlated branches. For example, we learn sequential rules from PT traces under normal executions for each identified coarse-grained correlated branch pair.
5.5 Experimental Validation

In this section, we evaluate Dedop’s detection capability against DOP attacks. We first conduct a comprehensive case study of the ProFTpd DOP attack, which is the only available DOP attack provided in [21]. Then, we conduct an empirical study of our branch correlation analysis tool to demonstrate the prevalence of correlated conditional branches. Finally, we report the runtime performance overhead of PT tracing in Dedop.

5.5.1 Detection of DOP attack

Experiment Setup

We have ported the original ProFTPD attack to the 32bit Ubuntu 16.04 setup. The attack heavily relies on the precise knowledge of library and memory layout. In particular, we modified the original metasploit module and created an automated script to scan the ProFTPD binary, which automatically locates the targeted memory addresses. Since the ProFTpd DOP attack runs on a 32-bit platform, we collected control-flow traces using the Linux perf on a commodity Intel Skylake machine with 32-bit Ubuntu OS and Intel PT support. We discuss practical deployment issues for PT-based detection in Section 6.

To detect incompatible branch behavior and micro-level control-flow frequency anomalies, we focus on tracing control-flow behaviors of the vulnerable sreplace function, which is enabled by perf’s filter configuration. We identify correlated branches using our LLVM-based branch correlation analysis tool and Z3 [154] solver, and derive the branch correlation rules to enforce the branch behavior integrity.

We utilize the sCFT model for the detection of micro-level frequency anomaly. We consider all control-transfers in each sreplace invocation as a behavior instance. The training data is composed of a number of sreplace execution traces collected by PT. Then, cluster analysis is used to learn distinct normal behaviors from a set of distributions [42, 40]. The key difference between our sCFT model and solutions in [42, 40] is that our method embeds ordering information of control-transfers in the model, while their solutions are not.

In detecting macro-level interaction frequency anomalies, we use LBNL-FTP-PKT [2] dataset as the baseline interaction behaviors between a client and an FTP server. It contains all incoming anonymous FTP connections to public FTP servers at the Lawrence Berkeley National Laboratory over a ten-day period, totally 21482 FTP connections. Each connection is considered as a behavior instance. We can easily extract FTP commands in each connection from the dataset. For a ProFTpd
server program under the DOP attack, we can derive FTP commands sent from clients by tracing
control transfers of the FTP command dispatcher function _dispatch. PT captures the control
transfers from _dispatch to different command handlers, e.g., core_cwd indicates that the
command CWD (i.e., change working directory) has been received.

We capture the statistical properties of FTP client-server interactions in terms of frequency dis-
tributions of n-step (n is set to 2) FTP command sequences, which is similar as the modeling for
micro-level control-transfers. However, such distribution-based profiling may incur a high analysis
overhead, due to a large number of dimensions/features. To address this challenge, we apply the
Principal Component Analysis (PCA) technique for dimension reduction. We then adopt x-means
clustering approach [149] to cluster all behavior instances in normal program traces, where the
center of each of the x-clusters represents a normal program execution context.

**Incompatible Branch Behavior Anomaly**

Code 5.1 shows the off-by-one vulnerability in sreplace function, which is exploited by the
ProFTPd DOP attack [21]. Our branch correlation analysis tool identifies 14 coarse-grained corre-
lated conditional branches in sreplace, including the conditional branches in lines 21 and 30 of
Code 5.1 since they are both data-dependent on the same variables cp, pbuf, and blen. Fig. 5.6
shows these two correlated branches, BR1 and BR2, respectively.

![Figure 5.6: Example of an incompatible branch behavior](image)

We use the Z3 [154] logic solver to identify any "subsume" or "mutually exclusive" relationship
between branch predicates. For example, to derive the one-way "subsume" relationship from BR2
to BR1, we first add the predicate in BR2 as a constraint into Z3 solver. However, as shown in
Code 5.1, cp gets redefined in line 31 and it is true-control-dependent on the branch in line 30.
In this case, we need to use the statement in line 31 as the constraint, i.e., cp==pbuf+blen-1.
Then, we add the constraint Not(((cp+rlen)-pbuf+1)>blen) to the solver, i.e., the logical
Not of the predicate in BR1. Since the variable rlen is the length of a non-null string (which can
be derived from the source code), we add \( rlen > 0 \) as an additional constraint. In this case, the solver returns \textit{unsat} (satisfiable), and thus we derive that BR2 "subsumes" BR1. That is, if BR2 returns \textit{true}, we can predict that BR1 should also take the \textit{true} branch. Finally, we obtain a one-way branch correlation rule, which will be enforced at runtime for incompatible branch behavior detection.

To trigger the memory corruption error in \texttt{sreplace}, the attacker first fills up \texttt{buf} and overwrites \texttt{buf}'s last character with a non-zero byte in line 33 of Code 5.1. The postfix increment operation \(*cp++\) makes the predicate in line 30 to return \textit{true} in the next iteration, and subsequently \texttt{cp} is reset to point to the last character of \texttt{buf} in line 31, as shown in \textcircled{1} in Fig. 5.6. Since \texttt{buf}'s last character is a non-zero value, it becomes a non-terminated string. As a result, \texttt{strlen(pbuf)}>\texttt{blen} (\textcircled{2} in Fig. 5.6), which enables the attacker to corrupt the local variables such as \texttt{cp} and \texttt{blen} in line 20. To bypass the overflow checking in lines 21-27, the attacker needs to make sure that the predicate in line 21 returns \textit{false}.

At runtime, DEEDOP observes a control-transfer from 0x805e4e2 in \texttt{sreplace} to 0x80638bf in \texttt{pr_log_pri}, which indicates that the predicate in line 21 has taken the \textit{true} branch. According to the derived branch correlation rule, it updates the branch status table that the predicate in line 30 is expected to take the \textit{true} branch in the following program execution. However, during the ProFTPd DOP attack, PT reports a control-transfer from 0x805e473 to 0x805e49a, which shows the predicate in line 30 takes the \textit{false} branch. As a result, DEEDOP detects this incompatible branch behavior and raises an alarm.

**Micro-Level Control-Flow Frequency Anomaly**

As mentioned in Section 5.2, changing the working directory (\textit{i.e.}, \texttt{CWD} command) triggers the invocation of \texttt{sreplace} function. The \texttt{.message} file is the input of \texttt{sreplace}, where its content determines the control-flow behaviors in \texttt{sreplace}. In ProFTPd DOP attack, an attacker needs to craft malicious payloads to repeatedly fill up the allocated buffer \texttt{buf} and write bytes beyond the buffer, which exhibits frequency anomalies of control-transfers in \texttt{sreplace}.

In this experiment, we randomly generate 1000 \texttt{.message} files without triggering the overflow as the baseline normal executions (\textit{i.e.}, the training dataset). Given a new behavior instance (\textit{i.e.}, PT traces) observed in the detection phase, we measure the distance between the new observation and each of the cluster centers in the training dataset. If it does not belong to any existing cluster (\textit{i.e.}, the distance is higher than a specified threshold of 30 unit length), it is considered an anomaly.

The ProFTPd DOP attack involves intensive interactions with the server, which triggers more than 180 invocations of the \texttt{sreplace} function. Fig. 5.7 shows an instance of the detection of
control-flow frequency anomaly in *sreplac*e. After applying PCA, we reduce the original high-dimensional data to 3-dimensional data and then perform the X-clustering. The training dataset has been clustered into 23 clusters. It shows that the distance between the testing instance and any existing cluster is abnormally large (the average distance is larger than 60-unit length in the three-dimensional space). Since the testing instance (*i.e.*, red triangle) does not belong to any existing cluster (*i.e.*, blue dots), it is considered an anomaly.

Note that incompatible branch behaviors may also result in frequency anomalies. Therefore, statistical modeling of control-transfers has the potential to detect not only the anomalous frequency of legitimate control-flow paths, but also incompatible branch behavior anomalies in data-oriented attacks.

**Interaction Frequency Anomaly**

We investigate the interaction frequency anomaly caused by ProFTPd DOP attack. Fig. 5.8 shows the 2-gram distribution of FTP commands within a connection during the ProFTPd DOP attack. It involves more than 1000 client-server FTP commands, while the average interactions per session in LBNL-FTP-PKT dataset [2] is 41.
Fig. 5.8: 2-gram distribution of FTP commands during the ProFTPd DOP attack

Fig. 5.9 illustrates the X-clustering for 2-grams of FTP commands with PCA reduction to 3-dimension. Similar to Fig. 5.7, the testing instance (i.e., red triangle) does not belong to any existing cluster (i.e., blue dots). The DOP attack involves an abnormally high number of client-server interactions. We can easily identify the DOP attack instance using interaction frequency anomaly detection with a specified threshold.

### 5.5.2 Detection of DDM Attack

Next, we provide case studies of detectable DDM attacks by control-flow tracing. Code 5.2 shows an example of the attack on a decision-making data in SSH server [1], where a local flag variable `authenticated` is used to indicate whether a remote user has passed the authentication (line 3). When the vulnerable function `packet_read()` is called (line 6), an attacker is able to corrupt the `authenticated` variable to a non-zero value, which always leads to a successful user authentication (line 16). In this example, the conditional branches in lines 5 and 16 are correlated, since they both are data-dependent on `authenticated`. We can detect such an attack with PT tracing after `authenticated` is corrupted at line 6, because of the incompatible branch behavior before and after the data manipulation site in line 6. The anomaly will be detected when the corrupted variable is used (i.e., in Stage S3 of the attack).

```c
void do_authentication(char *user, ...) {
```
Figure 5.9: X-clustering for 2-grams of FTP commands with PCA reduction to 3-dimension using LBNL-FTP-PKT dataset [2]. The DOP attack involves an abnormally high number of client-server interactions.

```c
... int authenticated = 0;
...
while(!authenticated){
    type = packet_read(); // manipulation site
    /* Corrupt the authenticated flag */
    switch(type){
        ...  
        case SSH_CMSG_AUTH_PASSWORD:
            if(auth_password(user, password)){
                authenticated = 1;
                break;}
        case ...
            if(authenticated) break;
    }  
    do_authenticated(pw);
    /* Perform session preparation */
}
```

Code 5.2: Code snippet in SSH server
For server-side applications with long-living decision-making variables, individual conditional branch may generate incompatible branch behaviors before and after the data manipulation in different rounds of client-server interaction. Take the example in Code 2.1, after the attacker overwrites `pw->pw_uid` with the root user’s id, the conditional branch behavior related to `pw->pw_uid` will be incompatible with the ones before the data corruption.

Though DOP attacks are expressive and potentially powerful [21], they may manifest unusual control-flow behavior in different dimensions and stages. From the above case study, as a complementary mitigation technique, we demonstrate the feasibility and effectiveness of using PT-based program anomaly detection to defeat DOP attacks. Since DOP attacks usually involve multiple-step data manipulations, they are less evasive from detection than DDM attacks.

### 5.5.3 Evaluation of Correlated Branch Identification

In many instances, data-oriented attacks require the arbitrary address write ability [1, 22, 23], where an attacker may tamper with any variable in the vulnerable program. In this sense, we conduct an empirical study on the prevalence of correlated conditional branches in a program, which reflects the possibility of data-oriented attacks to manifest incompatible branch behavior (in case if a non-control variable in a branch is manipulated).

We first obtain the LLVM Intermediate Representation (IR) of a program, and construct the program dependence graph, including both data and control dependencies. In addition to the direct data-dependency, we capture indirect (e.g., two branches using different predicate variables but having a joint data-dependency) and inter-procedural branch correlations. For each conditional branch instruction, we find a pair of correlated branches if it can be directly or indirectly linked to another conditional branch through the program dependence graph. We also count the number of simple forms of conditional predicates. When a branch instruction only has two backward data-dependent instructions (i.e., icmp and load), we consider it as a simple form of branch (referred to as simple BR in Table 5.1). If we can use the logic solver Z3 [154] to determine any "subsume" or "mutually exclusive" relationship between these correlated branches, we mark them as "statically derivable".

We choose 4 Linux utility programs (`flex`, `grep`, `gzip`, and `sed`) from the Software-artifact Infrastructure Repository (SIR) [95] and 4 vulnerable programs (`wu-ftp`, `orzhttpd`, `ghttpd`, and `sudo`) from the FlowStitch benchmarks [19] in our empirical study. Table 5.1 reports the results of coarse-grained branch correlation analysis. From the results, on average 77% of branches have at least one correlated branch (as shown in the third column of Table 5.1). 24% branches exhibit simple forms of conditional predicates. The average percentage of simple correlated branches is 18%, where we can easily use a satisfiability modulo theories (SMT) logic solver, e.g., the state-of-the art theorem prover Z3 [154], to determine any "subsume" or "mutually exclusive" relationship
Table 5.1: Branch correlation analysis

between branches. Our analysis also shows the limitation of the approach in [100], where only the simple forms of predicates are considered in finding correlated branches. Another observation is that, we can only derive a very limited number of correlations through the logic solver. To overcome this limitation, we may use data mining techniques to identify correlated branches given sufficient training data. However, there exists false positives in this approach.

5.5.4 Tracing Overhead

In Section 4.4.4, we have compared the tracing overhead of Linux perf (version 4.13 with PT support) and Strace (version 4.13 with stack unwinding support enabled) using the Linux utility program grep (809 test cases). To accurately measure the tracing overhead of the vulnerable function sreplace, we craft a self-contained model program that preserves the semantics of sreplace. We run sreplace for 10000 times, and compare the program execution time with system-call tracing and PT tracing, respectively. The baseline refers to the execution time without running any tracing tool. Our testing platform is a desktop computer with an Intel i7-6700K Skylake CPU @ 4.00GHz, running 32bit Ubuntu OS. All the results have been averaged over 10000 runs, and the related standard deviations are provided as error bars.

Fig. 5.10 shows the tracing overhead. We observe that perf for PT tracing incurs a low overhead with around 14% slowdown. Strace tracing incurs a rather significant runtime overhead around 357% slowdown. The size of raw trace data is around 0.03MByte. The low-overhead tracing by PT clearly increases the feasibility for practical deployment of program anomaly detection.
5.6 Summary

In this work, we focused our investigation on program anomaly detection against DOP attacks using PT tracing. DOP is a recently proposed advanced technique to construct non-control data exploits, which has received considerable attention due to its rich expressiveness. We demystified the DOP exploitation technique by dissecting the real-world ProFTPd DOP attack into steps. We observed that DOP attacks can potentially cause three types of anomalous control-flow behaviors. For the detection of DOP attacks, we proposed to enforce the branch correlation integrity and detect both micro and macro levels of frequency anomalies. To derive branch correlations in a program, we developed a general branch correlation analysis tool based on LLVM [47] to automatically identify coarse-grained correlated branches, and then used the logic solver to identify strong correlated branches. Through experimental validation, we demonstrated that DeDOP can successfully detect program behavior anomalies in different dimensions against the real-world ProFTPd DOP attack, and it can also be applied to detect basic non-control-data attacks.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, I presented attempts to detect data-oriented attacks through program anomaly detection techniques. Specifically, I made three technical contributions to the area of program anomaly detection against data-oriented attacks: 1) enforcing cyber-physical execution semantics to defend against data-oriented attacks in Cyber-Physical Systems (CPS); 2) statistical program behavior modeling for frequency anomaly detection; and 3) defending against the newly proposed data-oriented programming (DOP) attacks using Intel Processor Trace (PT).

In Chapter 3, I described a new security methodology for defending against data-oriented attacks by enforcing cyber-physical execution semantics in CPS. The key idea is to leverage the event-driven nature in characterizing CPS control program behaviors. The proposed method detects data-oriented exploits if a specific physical event is missing along with the corresponding event-dependent control-flow path. I also presented a general method for reasoning about cyber-physical execution semantics of a control program, including the event identification and dependence analysis. I implemented a proof-of-concept prototype on Raspberry Pi platforms, and evaluated the prototype’s performance by conducting case studies under data-oriented attacks.

In Chapter 4, I presented two statistical program behavior models: 1) $s$FSA at the system-call level; and 2) $s$CFT at the control-flow-transfer level (where the control-flow tracing is offered by Intel PT). The statistical program behavior modeling captures aberrant occurrence frequencies hidden in long-distance system events (i.e., system-calls or control-transfers), but also preserves local temporal relations among adjacent system events. I implemented prototypes of the proposed approaches and conducted an extensive experimental evaluation to demonstrate their effectiveness against real-world data-oriented attacks and synthetic data-oriented anomalies.
In Chapter 5, I analyzed the real-world ProFTPd DOP attack to demystify the DOP exploitation technique. I designed and implemented the DeDOP anomaly detection system. DeDOP enforces the branch correlation integrity and improves anomaly detection sensitivity by capturing statistical characteristics of both short control-transfers and client-server interactions. In particular, I developed a general branch correlation analysis tool to automatically identify coarse-grained correlated branches. Through experimental validation, I showed that DeDOP can successfully detect anomalies in different dimensions against the ProFTPd DOP attack.

6.2 Future Work

Program anomaly detection can be considered a complementary defense mechanism to in-lined preventative defenses against cyber attacks. There are many research issues and opportunities where further research efforts are required in this area. An important research direction is securing CPS against malicious attacks using program anomaly detection. The second line of research is about exploitation techniques of data-oriented attacks and the corresponding defenses. I highlight few of them in the following.

6.2.1 Program Anomaly Detection in CPS

As CPS are increasingly used to operate critical infrastructures and quickly evolving from isolated devices to interconnected computing and control units, CPS devices are exposed to a large attack surface, and vulnerable to both cyber and physical attacks. As the interfaces between cyber and physical components, control programs running on either field devices or the control center play an important role in CPS. This also leads control programs to be major targets of CPS attacks. In addition to software exploits, random hardware memory errors may corrupt decision-making data and cause devastating consequences equivalent to CPS attacks. Program anomaly detection approaches are effective in uncovering such anomalous CPS behaviors.

The most distinctive feature of CPS from other IT systems is the interaction between control programs and the physical space. To achieve attack goals, it is likely the attacks need to cause an adverse effect on the physical system that will not match with the expected CPS behavior [105]. The feature of cyber-operations with physical dependencies and consequences in CPS can be used for mitigating potential attacks and securing the control systems. The causal dependencies between program behavior and the physical environment have been less studied in the literature, which, however, is of great help to identify advanced CPS attacks and anomalies.

Moving forward, I plan to design the Event Triggering and Control Actuation Integrity (ETCAI) se-
security enforcement, which aims to check inconsistencies between the physical context and program execution for CPS with finer granularity and in a preventative manner. ETCAI targets at detecting two general types of advanced attacks in CPS: 1) adversary event triggering attacks, which can be caused by either data-oriented attacks, control-oriented attacks, or hardware-based attacks; and 2) control command replacing attacks [38], which could replace legitimate control commands with malicious ones before they are sent out to the physical plant’s actuators.

One possible approach can be based on the compile-time code instrumentation and external runtime program execution monitoring. Specifically, we insert signalling hooks at the entry and exit points of any sensitive control action. The program sends a signal to the execution monitor before and after executing any critical action, and then the monitor verifies that the program execution is consistent with the corresponding physical context in terms of cyber-physical dependencies and cyber-physical consequences. Before a sensitive action is actuated, we check whether the current physical context indicates that the upcoming control action should be executed (i.e., event triggering integrity). After the control program triggers a validated action, we check whether the control action has properly happened (i.e., control actuation integrity). My design enables an anomaly detection system to identify adversary event triggering before a critical action is taken at runtime, thus effectively preventing attacks from damaging the physical system.

6.2.2 Data-Oriented Attacks and Mitigation

Automated Evasive DOP Attack

The construction of a DOP attack still involves a lot of manual efforts, from gadget preparation, exploit chain construction, to stitchability verification [21], which require non-trivial engineering efforts and impose difficulties for attackers. Attackers need to have an in-depth application-specific and vulnerability-specific knowledge of the target program. Future research efforts are needed to lower the bar for constructing DOP exploits, particularly the non-interactive DOP attack.

As shown in Section 5.5, a DOP attack often manifests itself in terms of unusual control-flow behaviors. One interesting research topic is how to construct evasive DOP attacks in the presence of state-of-the-art defenses. For example, Hu et al. [21] demonstrated the prevalence of data-oriented gadgets in programs. Our empirical study shows that on average 43% gadgets are involved in at least one conditional branch using vulnerable programs from the FlowStitch benchmarks [19]. These gadgets may potentially have impacts on control-flow behaviors if used for DOP. Attackers may strategically choose data-oriented gadgets that incur minimum unusual control-flow behaviors to evade detections.
Efficient PT-based Detection

Though hardware-assisted program execution tracing such as PT has the potential to bring anomaly detection closer to reality, it is impractical to trace the complete control-flow transfers of a program at runtime due to performance slowdown and storage overhead constraints. Given a limited overhead budget, it is important to determine strategic checkpoints (e.g., setting filters to monitor potentially vulnerable functions) for efficient online monitoring. As I discussed in Section 2.3.3, there are several undetectable cases by control-flow tracing. Such data-oriented attacks may have a correlation with other hardware events. An interesting direction is to explore new hardware events for anomaly detection against data-oriented attacks.

Deep Learning for Branch Behavior Modeling

Deep learning techniques have shown promises in detecting anomalies in different applications [155]. Recent work [156] presents a deep learning approach to classify PT generated control-flow traces for malware detection. An important research direction is to apply deep learning algorithms to model program behaviors for anomaly detection. For example, manipulated non-control data may have impacts on control-flows in different locations with long distances in a program. Long Short-Term Memory (LSTM) is able to keep track of temporally distant events, and thus has the potential to detect incompatible control-flow paths given an extremely long control-flow sequence.
Bibliography


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


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


