Exploitation Techniques for Data-Oriented Attacks with Existing and Potential Defense Approaches*

LONG CHENG, School of Computing, Clemson University, USA
SALMAN AHMED, Department of Computer Science, Virginia Tech, USA
HANS LILJESTRAND, David R. Cheriton School of Computer Science, University of Waterloo, Canada
THOMAS NYMAN, Department of Computer Science, Aalto University, Finland
HAIPENG CAI, School of Electrical Engineering and Computer Science, Washington State University, USA
TRENT JAEGER, Department of Computer Science and Engineering, Pennsylvania State University, USA
N. ASOKAN, David R. Cheriton School of Computer Science, University of Waterloo, Canada
DANFENG (DAPHNE) YAO, Department of Computer Science, Virginia Tech, USA

Data-oriented attacks manipulate non-control data to alter a program's benign behavior without violating its control-flow integrity. It has been shown that such attacks can cause significant damage even in the presence of control-flow defense mechanisms. However, these threats have not been adequately addressed. In this survey paper, we first map data-oriented exploits, including Data-Oriented Programming (DOP) and Block-Oriented Programming (BOP) attacks, to their assumptions/requirements and attack capabilities. Then, we compare known defenses against these attacks, in terms of approach, detection capabilities, overhead, and compatibility. It is generally believed that control flows may not be useful for data-oriented security. However, data-oriented attacks (especially DOP attacks) may generate side effects on control-flow behaviors in multiple dimensions (*i.e.*, incompatible branch behaviors and frequency anomalies). We also characterize control-flow anomalies caused by data-oriented attacks. In the end, we discuss challenges for building deployable data-oriented defenses and open research questions.

CCS Concepts: • **Security and privacy → Systems security**; *Software security engineering*; *Intrusion/anomaly detection and malware mitigation.*

Additional Key Words and Phrases: data-oriented attacks, DOP, BOP, branch correlation, frequency anomalies

ACM Reference Format:
Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. 2018. Exploitation Techniques for Data-Oriented Attacks with Existing and

* A preliminary version of the work appeared in Cheng, L., Liljestrand, H., Ahmed, M. S., Nyman, T., Jaeger, T., Asokan, N., & Yao, D. (2019, September). Exploitation techniques and defenses for data-oriented attacks. In 2019 IEEE Cybersecurity Development (SecDev) (pp. 114-128). IEEE.

Authors' addresses: Long Cheng, lcheng2@clemson.edu, School of Computing, Clemson University, USA; Salman Ahmed, ahmedms@vt.edu, Department of Computer Science, Virginia Tech, USA; Hans Liljestrand, hans@liljestrand.dev, David R. Cheriton School of Computer Science, University of Waterloo, Canada; Thomas Nyman, thomas.nyman@aalto.fi, Department of Computer Science, Aalto University, Finland; Haipeng Cai, haipeng.cai@wsu.edu, School of Electrical Engineering and Computer Science, Washington State University, USA; Trent Jaeger, trj1@psu.edu, Department of Computer Science and Engineering, Pennsylvania State University, USA; N. Asokan, asokan@acm.org, David R. Cheriton School of Computer Science, University of Waterloo, Canada; Danfeng (Daphne) Yao, danfeng@vt.edu, Department of Computer Science, Virginia Tech, USA.

0004-5411/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

Potential Defense Approaches*. *J. ACM* 37, 4, Article 111 (August 2018), 35 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Memory-corruption vulnerabilities are one of the most common attack vectors used to compromise computer systems. Attackers exploit these vulnerabilities in different ways to perform arbitrary code execution and data manipulation. Existing memory corruption attacks are broadly two types: *i)* control-flow attacks [41, 87, 95] and *ii)* data-oriented attacks (also known as non-control attacks) [18, 49, 51, 73, 105]. Both types of attacks can cause significant damages to a victim system [14, 38].

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 [41], code reuse [82], and Return-Oriented Programming (ROP) [95]. Defenses such as stack canaries [25], Data Execution Prevention (DEP) [5], Address Space Layout Randomization (ASLR) [107], Control-Flow Integrity (CFI) [1], Intel's CET [52] and MPX [53] can prevent most control-flow attacks. In particular, CFI-based defenses [12, 43, 80, 123] have received considerable attention in the last decade. The idea is to ensure that the runtime program execution always follows a valid path in the program's Control-Flow Graph (CFG), by enforcing policies on indirect control transfer instructions (*e.g.*, ret/jmp).

In contrast to control-flow attacks, data-oriented attacks [18] change a program's benign behavior by manipulating the program's non-control data (*e.g.*, a data variable/pointer which does not contain the target address for a control transfer) without violating its control-flow integrity. The attack objectives include: 1) information disclosure (*e.g.*, leaking passwords, private keys or address space layout); 2) privilege escalation (*e.g.*, by manipulating user identity data) [18]; 3) performance degradation (*e.g.*, resource wastage attack) [7]; and 4) bypassing security mitigation mechanisms [119]. As launching control-flow attacks becomes increasingly difficult due to many deployed defenses, data-oriented attacks have received much attention in the literature [49, 51, 73, 77, 90, 119].

Data-oriented attacks can be as simple as flipping a bit of a variable. However, they can be equally powerful and effective as control-flow attacks [49, 54]. For example, arbitrary code-execution attacks are possible if an attacker can corrupt parameters of system calls (*e.g.*, `execve()`) [14]. Recently, Hu *et al.* [51] have proposed Data-Oriented Programming (DOP), a systematic technique to construct expressive (*i.e.*, Turing-complete) non-control data exploits. Ispoglou *et al.* [54] also presented a code-reuse technique called Block-Oriented Programming (BOP) that utilizes basic blocks as gadgets along valid execution paths in the target binary to generate data-oriented exploits. Though data-oriented attacks have been known for a long time, the threats posed by them have not been adequately addressed due to the fact that most previous defense mechanisms focus on preventing control-flow exploits.

The motivation of this paper is to systematize the current knowledge about exploitation techniques of data-oriented attacks and the current applicable defense mechanisms. Unlike prior papers [62, 103, 105] related to memory corruption vulnerabilities, our work specifically focuses on data-oriented attacks. In addition to generic memory corruption prevention mechanisms discussed in [62, 103, 105] such as memory safety, software compartmentalization, and address/code space randomization, we mainly discuss recently proposed defenses against data-oriented attacks. Our technical contributions are as follows.

* We systematize and categorize the current knowledge about data-oriented exploitation techniques with a focus on the recent DOP attacks. We demystify the DOP exploitation technique by using the ProFTPd DOP attack [49] as a case study, and provide an intuitive and detailed explanation of this attack by analyzing its constituent steps. We discuss the automation of data-oriented attacks,

e.g., the Block-Oriented Programming (BOP) compiler [54], STEROIDS [83], and LIMBO [92]. We also discuss representative data-oriented exploits including their assumptions/requirements and attack capabilities (Section 2).

* We present a three-stage model for data-oriented attacks and discuss recent defense techniques according to different stages. Then, we provide a comparative analysis of the approaches focusing on data-oriented attacks (Section 3).
* We characterize the side effects of DOP attacks on control-flow behaviors in multiple dimensions in terms of the branch correlation and frequency distribution of control-flow transfers. Our results show evidence of incompatible branch behavior and frequency anomalies caused by DOP attacks (Section 4). Such preliminary characterization efforts could be starting points for building potential defenses in the future.
* We also discuss some open research problems and unsolved challenges (Section 5).

2 DATA-ORIENTED ATTACKS

In this section, we first describe why data-oriented attacks have received attention among security researchers in recent years (Section 2.1). Then we discuss two categories of data-oriented exploitation techniques (Section 2.2) and automatically generating data-oriented exploits (Section 2.3). We reproduce a real-world DOP attack against the ProFTPD FTP-server[1] [51] with a detailed description of the attack to demonstrate its rich expressiveness in Section 2.4. We also provide a brief overview of the BOP attack technique in Section 2.5. We discuss the similarity and difference between DOP and BOP and the generality and practicality (difficulty) of both DOP and BOP exploits in Section 2.6. Then, we map representative data-oriented exploits in the literature to their assumptions/requirements and capabilities (Section 2.7).

2.1 Why do data-oriented attacks receive attention?

The Data Execution Prevention (DEP) or No-Execute (NX) defense prevents stack smashing [79] and related attacks such as overwriting Structured Exception Handler. A new class of attacks, namely the code-reuse attacks such as ROP [95], dominated in the last decade due to their capability of bypassing DEP or NX. However, researchers have put significant effort to develop practical security solutions for preventing code-reuse attacks. The solutions are broadly in five categories: *i)* fine-grained address space randomization (ASR [44], ASLP [56], CCR [57], Selfrando [23], etc.), *ii)* re-randomization (*e.g.*, TASR [10], Shuffler [117], Remix [19], ASLR-Guard [69], etc.), *iii)* memory leakage prevention (*e.g.*, ASLR-Guard [69], XnR [6], Readactor [27], Heisenbyte [106], etc.), *iv)* code pointer integrity (*e.g.*, CPI [59], PointGuard [26], etc.), and *v)* CFI (*e.g.*, BCFT [43], CCFIR [123], bin-CFI [125], etc.). The enforcement of the above defenses makes most code-reuse attacks unreliable. Thus, many attackers have shifted their focus from control-oriented attacks to data-oriented attacks in recent years [51, 54].

2.2 Classification of data-oriented attacks

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

1) DDM refers to a category of attacks in which an attacker directly/straightforwardly 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 or offset to a known location utilized in the attack can be derived directly from binary analysis (*e.g.*, global variable with a deterministic address)

[1]The automated scripts of launching the ProFTPd DOP attack are provided at https://github.com/doppt.

or by reusing the runtime randomized address stored in memory [49]. Several types of memory corruption vulnerabilities, *e.g.*, format string vulnerabilities, buffer overflows, integer overflows, and double free vulnerabilities [103], allow attackers to directly overwrite memory locations within the address space of a vulnerable application.

```
1  void do_authentication(char *user, ...) {
2      ...
3      int authenticated = 0;
4      ...
5      while(!authenticated){
6          type = packet_read();//Corrupt authenticated
7          /*Calls detect_attack() internally*/
8          switch(type){
9              ...
10             case SSH_CMSG_AUTH_PASSWORD:
11                 if(auth_password(user, password)){
12                     authenticated = 1;
13                     break;}
14             case ...
15         }
16         if(authenticated) break;
17     }
18     do_authenticated(pw);
19     /*Perform session preparation*/
20 }
```

Listing 1. DDM attack in a vulnerable SSH server [18]

Chen *et al.* [18] revealed that DDM attacks can corrupt a variety of security-critical 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. Listing 1 illustrates an of attacking decision-making data in the SSH server, first reported in [18]. A local flag variable `authenticated` indicates whether a remote user has passed the authentication (line 3). An integer overflow vulnerability exists in the `detect_attack()` function, which is internally invoked whenever the `packet_read()` function is called (line 6). When the vulnerable function is invoked, an attacker can corrupt the `authenticated` variable to a non-zero value, which bypasses the user authentication (line 16). DDM allows attackers to manipulate the benign data flows in a program execution without changing its control flow. FlowStitch [49] is a technique to stitch data flows in a vulnerable program to automatically construct data-oriented exploits. It takes a pair of source (*e.g.*, private key buffer) and target (*e.g.*, a publish output buffer) in a vulnerable program as the input. The goal is to stitch the source data-flow to the target data-flow, which could leak passwords, private keys, or randomized values, and cause privilege escalation.

There also exists multi-step DDM attacks, where an adversary exploits memory corruption vulnerabilities multiple times to write data to adversary-chosen memory locations. For example, suppose an attacker needs to change two decision-making variables while the vulnerability only allows the attacker to change one value each time. It requires a 2-step DDM. Morton *et al.* [73] recently demonstrated a multi-step DDM with Nginx (listed in Table 1). The attack leverages memory errors to modify global configuration data structures in web servers. Constructing a faux SSL Config struct in Nginx requires as many as 16 connections (*i.e.*, 16-step DDM) [73].

Due to the widespread deployment of ASLR, attackers may exploit DDM to infer knowledge about the address space layout of a process to bypass ASLR defenses. Data manipulation can also

cause some events that result in software side-channels—typically timing side-channels—that can leak information about the address space. To infer information about an application's address space, attackers analyze the output and execution time of a portion of code. They then correlate the output and timing information by running the same portion of code locally [37, 93]. In contrast to traditional DDM attacks with direct malicious goals, *e.g.*, leaking sensitive data or modifying program behavior, inferring randomized addresses serves as the first step towards malicious goals. For example, attackers need to derandomize the address space layout before performing code reuse attacks using DOP gadgets. For example, an attacker may overwrite a data pointer (`ptr` in Listing 2) to point to some byte sequences and infer knowledge about the byte sequences by observing the output, *i.e.*, the variable `result`. By pointing the data pointer to different locations of an address space and observing the output behavior, an attacker can distinguish the mapped and unmapped code pages.

```
1  struct mystruct {
2      int value;
3  };
4  void vuln_function()
5  {
6      char buf[64];
7      int result=0, length, input;
8      struct mystruct * ptr;
9      recv(socket, buf, input);
10     ptr->value = strlen(buf);
11     while (result < ptr->value) result++;
12     send(socket, &result, length);
13 }
```

Listing 2. Data pointer manipulation to infer knowledge about address space layout. This figure is adopted from [93].

2) DOP constructs expressive data-oriented exploits [51]. It allows an attacker to perform arbitrary computations in program memory by chaining the execution of short instruction sequences (referred to as *DOP gadgets*). DOP gadgets are similar to ROP gadgets that can perform arithmetic/logical, assignment, load, store, jump, and conditional jump operations. However, unlike ROP gadgets, the execution of DOP gadgets follows valid paths in a CFG. We consider Block-Oriented Programming (BOP) [54] as a form of DOP because BOP also constructs exploits using a set of gadgets (details are in Section 2.5). However, instead of using short instruction sequences as gadgets, BOP [54] constructs exploits by chaining basic blocks as BOP gadgets. Both DOP and BOP adhere to CFI. Without loss of generality, we use DOP to represent this exploitation technique, which misinterprets multiple gadgets and chains these gadgets together by one or more dispatchers to achieve the desired outcome. A dispatcher is a fragment of logic that chain gadgets. A typical example of a dispatcher is a loop within the influence of a memory corruption vulnerability.

Typically, a DOP attack corrupts several memory locations in a program and involves multiple steps. To understand the complexity and the expressiveness of the DOP technique, we dissect a real-world DOP attack in Section 2.4. We differentiate the DOP exploitation technique from a multi-step DDM attack. 1) *Gadgets and code reuse.* Both DOP and BOP attack techniques involve reusing code execution through CFI-compatible gadgets. Multi-step DDM hinges on direct memory writes and does not involve any gadget executions. 2) *Stitching mechanism and ordering constraint.* In DOP and BOP attacks, how to orderly stitch gadgets to form a meaningful attack is important.

Multi-step DDM attacks, *e.g.*, crafting and sending multiple attack payloads to manipulate memory values, do not need any special stitching mechanism and thus there is no ordering constraint.

2.3 Automatically generating data-oriented exploits

Research efforts have been undertaken to automate the process of generating data-oriented exploits. However, identifying how to corrupt memory values for a successful data-oriented exploit is non-trivial due to the large space of memory state configurations and attackers cannot inject malicious code of their choice. FlowStitich [49] is a tool to automatically construct DDM from memory errors. It identifies the influence range of the memory errors from the error-exhibiting trace (by triggering memory errors) and generates constraints on the program input to reach memory errors. FlowStitich then performs data-flow analysis and security-sensitive data (*e.g.*, system call parameters or configuration data) identification using benign traces, and selects stitch candidates from the identified security-sensitive data flows. It finally checks the feasibility of creating new edges with the memory errors and produces the input needed to mount a data-oriented attack. Figure 1(a) presents an example of a web server `wu-ftpd` with the format string vulnerability (skipped on line 5). Figure 1(b) shows the corresponding two-dimensional data-flow graph (2D-DFG), where numbers on the time-axis are the line numbers in Figure 1(a). The `seteuid(pw->pw uid)` on line 9 is intended to drop the process' root privilege. Suppose attackers want to retain root privilege by exploiting the format string vulnerability on line 5. One straightforward approach, demonstrated by Chen *et al.* [18], is to use DDM to directly modify the value of `pw_uid`. In FlowStitich, the target flow `pw->pw_uid` is first identified as a security-sensitive data flow. Then, it automatically finds another source data-flow and stitches the target flow to the source flow to achieve the same attack goal. As illustrated in Figure 1(b), attackers may change the base pointer `pw` to an address of a structure with a constant 0 at the offset corresponding to the `pw_uid`. The vulnerable code then reads 0 and uses it as the argument of `seteuid`, achieving a privilege escalation attack.

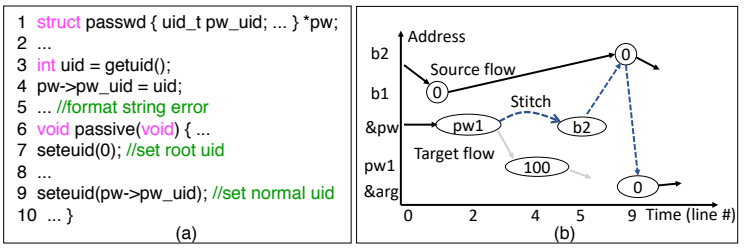

Fig. 1. Example of FlowStitch [49].

STEROIDS [83] is a compiler which automates the process of payload preparation in a DOP exploit, but which leaves the gadget search and DOP instance setup unaddressed. However, it is unclear how practical STEROIDS is for automating the construction of end-to-end DOP exploits since it assumes DOP gadgets are arbitrarily stitchable and work for all inputs. Constructing DOP or BOP exploits still requires a lot of manual work. Ispoglou *et al.* [54] presented a Block Oriented Programming Compiler (BOPC), a mechanism to automatically evaluate a program's remaining attack surface under strong control-flow hijacking mitigations. BOPC provides an exploit programming language, called SPL, that enables defenders and software developers to define exploits independent of the target program or underlying architecture. BOPC assumes that the target binary has an arbitrary memory write vulnerability. It finds basic blocks from the target program that implement individual SPL statements, and chains these basic blocks together. Finally, BOPC simulates the BOP chain to produce a payload that implements the SPL payload.

LIMBO [92] maps the automatic exploit generation problem into the software model checking problem. Similar to software model checking, LIMBO looks for possible state transitions from

an input state to the goal state, *i.e.*, the reachability from an input state to the goal state. To find the reachability, LIMBO employs concolic execution [16] with heuristics. The heuristics make the search space small by focusing on the promising paths that likely lead to the goal states. Similar to the BOPC [54], LIMBO [92] allows users to specify the input and goal states. In case users are unable to find vulnerabilities for an input state, LIMBO provides an option to make a synthetic buffer overflow by calling a function that triggers a stack buffer overflow. The goal expressions can be setting a register (*e.g.*, %ebx = 0x0804a010), writing to a memory (*e.g.*, Memory[0x0804842f] = 0xB), reading from a memory to a register (*e.g.*, %eax = Memory[0x0804a018]) or executing commands (*e.g.*, system("/bin/sh")). Both BOPC and LIMBO use concolic execution and heuristics to make state transitions. However, the key difference between these two techniques is how the two techniques look for the goal state. BOPC divides an exploit goal into several smaller goals, confirms the reachability of the smaller goals, and combines the smaller goals to achieve the targeted goal. On the other hand, LIMBO considers the exploit goal as a single state and searches for the exploiting state through the concolic execution.

The difficulty of attack generation using data-oriented exploit generation tools depends on whether the tools support the end-to-end exploit generation. If the exploit generation tool does not generate end-to-end exploits, then attackers must manually set up the initial phase of an exploit, *e.g.*, finding vulnerabilities to control memory. Also, attackers must interpret the output of the automatic exploit generation tools. The interpretation of a tool's output can be straightforward. However, the setup of the initial phase can be challenging and varies between tools. For example, BOPC requires arbitrary memory read/write primitives and an entry point for the initial phase. Similarly, LIMBO requires control over a program state by triggering vulnerabilities (*e.g.*, triggering stack buffer overflow to control a stack frame). On the other hand, STEROIDS requires the gadget lookup and gadget stitching methodologies to be provided by attackers. STEROIDS also requires the attackers to trigger a memory corruption vulnerability. Thus, in most cases, the initial phase of automatic exploit generation tools requires the discovery and triggering of vulnerabilities to control and leak memory.

Address space layout randomization can exacerbate the setup of the initial phase. In the presence of coarse-grained ASLR [107], attackers require one or more vulnerabilities to control a program's flow as well as to leak memory. The memory leak is necessary for obtaining knowledge about the address space of a program. Moreover, a single memory leak is not sufficient in the presence of fine-grained ASLR (ASR [44], ASLP [56], CCR [57], Selfrando [23], etc.).

2.4 Demystifying the ProFTPd DOP attack

We use the ProFTPd DOP attack crafted by Hu *et al.* [51] to illustrate the typical flow of DOP attacks. The goal of this DOP attack is to bypass randomization defenses (such as ASLR [107]), and then leak the server's OpenSSL private key. The private key is stored on the heap with an unpredictable layout, which prevents the attacker from reliably reading out the private key directly. Though the key is stored in a randomized memory region, it can be accessed via a chain of 8 pointers. As long as the base pointer is not randomized, *e.g.*, when the Position Independent Executables (PIE) feature is disabled, it is possible to exfiltrate the private key by starting from the OpenSSL context base pointer (*i.e.*, a known location of the static variable `ssl_ctx`) and recursively dereferencing 7 times within the server's memory space.

2.4.1 The ProFTPd vulnerability. ProFTPD versions 1.2 and 1.3 have a stack-based buffer overflow vulnerability in the `sreplace` function (CVE-2006-5815 [76]). The overflow can be exploited to obtain an arbitrary write primitive. The server program provides a feature to display customized messages when a user enters a directory. The message content is saved in `.message` file in each

directory. It can be edited by any user with write-access to the directory. The `.message` file can contain special characters (*i.e.*, specifiers) 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 the `main_server→ServerName` string, and "%T" will be replaced by the current time and date. Changing the working directory with a `CWD` command triggers the processing of `.message`, and subsequently triggers the invocation of the `sreplace` function. To trigger a memory error in `sreplace`, the attacker prepares payloads in the `.message` files, and then sends `CWD` commands to the server.

```
1  char *sstrncpy(char *dest, const char *src, size_t n) {
2      register char *d = dest;
3      for (; *src && n > 1; n--)
4          *d++ = *src++;
5      ...
6  }
7  char *sreplace(char *s, ...) {
8      ...
9      char *m,*r,*src = s,*cp;
10     char **mptr,**rptr;
11     char *marr[33],*rarr[33];
12     char buf[BUF_MAX] = {'\0'}, *pbuf = NULL;
13     size_t mlen=0, rlen=0, blen; cp=buf;
14     ...
15     while(*src){
16         for(mptr=marr, rptr=rarr; *mptr; mptr++, rptr++) {
17             mlen = strlen(*mptr);
18             rlen = strlen(*rptr);
19             if(strncmp(src,*mptr,mlen)==0){//check specifiers
20                 sstrncpy(cp,*rptr,blen-strlen(pbuf)); //replace a specifier
       with dynamic content stored in *rptr
21                 if(((cp + rlen) - pbuf + 1) > blen){
22                     cp = pbuf + blen - 1; ...
23                 } /*Overflow Check*/
24                 ...
25                 src += mlen;
26                 break;
27             }
28         }
29         if(!*mptr) {
30             if((cp - pbuf + 1) > blen){ //off-by-one error
31                 cp = pbuf + blen - 1; ...
32             } /*Overflow Check*/
33             *cp++ = *src++;
34         }
35     }
36 }
```

Listing 3. The vulnerable function in ProFTPd

Listing 3 shows the vulnerable `sreplace` function. The vulnerability is introduced by an off-by-one comparison bug in line 30, and allows attackers to modify the program memory. A defective overflow check in lines 29-34 is performed to detect any attempt to write outside 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 string terminator is overwritten in line 33. Consequently, 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 `blen - strlen(pbuf)` of the `sstrncpy` function becomes negative, which will be interpreted as a large unsigned integer (in line 20). Hence, the invocation of `sstrncpy` overflows outside buffer bounds into the stack and overwrites local variables such as `cp`.

Both the source (*i.e.*, `*rptr`) and the destination (*i.e.*, `cp`) of the string copy function, *i.e.*, `sstrncpy` in line 20, are under the control of the attacker, where `*rptr` can be manipulated by the attacker through specifying special characters in `.message` (*e.g.*, "%C" will be replaced by an attacker-specified directory name). As a result, the vulnerability allows the attacker to control the source, destination, and number of bytes copied on subsequent iterations of the `while` loop in lines 15-35.

2.4.2 The attack flow. The attacker interacts with the server (over the course of numerous FTP commands) to corrupt program memory by repeatedly exploiting the buffer overflow vulnerability. In this scenario, the command handler `cmd_loop` in ProFTPd serves as the data-oriented gadget dispatcher. On each iteration, the attacker triggers the execution of targeted gadgets by sending a crafted attack payload to the server program, *e.g.*, the dereference gadget `*d++ = *src++` located in `sstrncpy` (line 4 in Listing 3). We reproduced the ProFTPd DOP attack, and observed that the vulnerable function `sreplace` is called over 180 times during the attack.

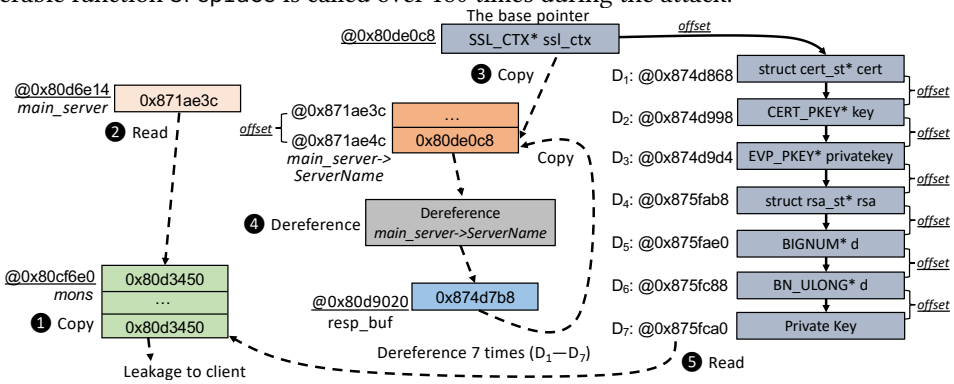

Fig. 2. ProFTPd DOP attack flow. An attacker needs to know the underlined addresses and offsets to launch the attack.

Figure 2 shows a step-by-step description of the ProFTPd DOP attack. The underlined addresses and offsets are acquired through binary analysis before launching the attack. During the attack, program memory is systematically corrupted to construct a DOP program out of individual operations. The main steps, shown in Figure 2, are described as follows.

❶ To read data from arbitrary addresses in the server, the 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 Figure 2, the `mons` array is filled with `G_PTR`'s address `0x80d3450`. Thus, when the server returns the date information to the client, it prints the value pointed by `G_PTR`. This step builds an exfiltration channel which can leak information from the server to the network.

❷ The attacker knows the memory address of the global pointer `main_server` at `0x80d6e14`, and reads the main server 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.

❸ The attacker knows the offset, 0x10, of the ServerName field in the main server structure and 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.

❹ The dereferencing operation dereferences the value located at main_server→ServerName, by triggering the execution of the dereference gadget in line 4 of Listing 3. The dereferenced value will be copied to a known position in the response buffer resp_buf. The attacker then obtains the address 0x874d868 of cert (D_1 in Figure 2) by adding the offset 0xb0 to the dereferenced value 0x874d7b8. After that, the attacker copies the address of cert to main_server→ServerName for the next iteration of deference. This step repeats 7 times (D_1~D_7 in Figure 2) following the dereference chain as shown in Figure 2. Finally, the address of the private key is obtained.

❺ The attacker sequentially reads 8 bytes from the private key buffer via the information exfiltration channel constructed in the first step. This process repeats for 64 times to retrieve a total of 512 bytes data.

2.5 Block-Oriented Programming (BOP) attack

Unlike DOP, BOP [54] constructs data-oriented exploits by chaining the basic blocks together instead of instructions. The core of BOP is the Block-Oriented Programming Compiler (BOPC). BOPC searches for the necessary basic blocks for an exploit. An exploit is written in a high-level C like language called SPloit Language (SPL) (Figure 3(a)). BOPC maps each SPL statement to a functional block and chains the functional blocks using a set of dispatcher blocks. The single- and double-bordered rectangles in Figure 3(b) represent dispatcher and functional blocks. A functional block executes the semantics of an SPL statement whereas a set of dispatcher blocks links between two functional blocks.

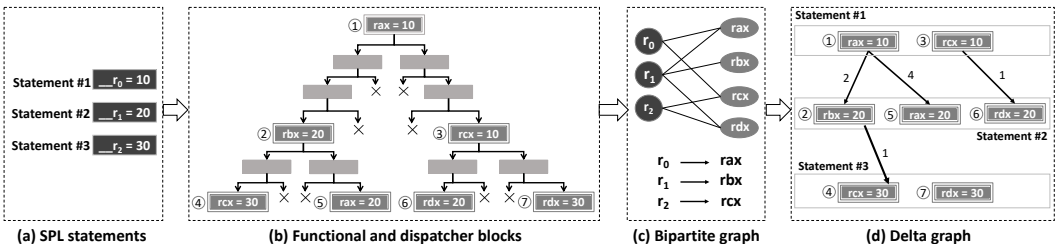

Fig. 3. Four major components of a BOP Compiler. The double and single border boxes (□) indicate functional and dispatcher blocks. The number inside a circle (◯) represents the functional block number. The ✕ represents irrelevant basic blocks.

To search and select functional blocks for SPL statements, BOPC creates a bipartite graph by associating each SPL statement with a set of functional blocks that may potentially serve the SPL statement. Figure 3(c) shows a bipartite graph for the SPL statements in Figure 3(a) where functional blocks ① and ③ can serve SPL statement #1; functional blocks ②, ⑤, and ⑥ can serve SPL statement #2; and functional blocks ④ and ⑦ can serve statement #3. BOPC selects an association from many possible associations. Figure 3(c) shows one such association ($r_0 \rightarrow rax$, $r_1 \rightarrow rbx$, and $r_2 \rightarrow rcx$).

BOPC selects a set of dispatcher blocks by constructing a delta graph as an arbitrary selection of the dispatcher blocks may also clobber the SPL state. A delta graph is a multipartite graph that has functional blocks as nodes. An edge of the graph represents the basic blocks that are necessary

for moving from one functional block to the next one with the numbers of basic blocks as edge weights. Figure 3(d) shows a delta graph for the SPL statements in Figure 3(a). A recursive version of Dijkstra's [24] shortest path algorithm minimizes the set of basic blocks required for moving from one functional block and another. This minimization process produces a sub-graph of the delta graph indicated by bold edges in Figure 3(d).

Once the subgraph is created, BOPC selects a functional block and translates the basic blocks between the selected and the next functional blocks into constraints by leveraging concolic execution [91]. Once the last functional block is reached, BOPC checks for satisfying assignments for these constraints. If the satisfying assignments are possible, BOPC produces a BOP gadget chain. BOPC does not produce an end-to-end exploit program. BOPC requires an entry point where the execution of the exploit payload should start and outputs a set of "what-where" memory writes to indicate the memory initialization and memory setup to execute the exploit payload.

2.6 Comparison between DOP and BOP

Both DOP and BOP are data-oriented attacks. However, DOP is focused more on the generalization side of data-oriented attacks, whereas BOP is focused on the automation side of data-oriented attacks. Thus, BOP has a few key differences from DOP in terms of *i)* automatic exploit generation, *ii)* exploit writability and *iii)* granularity. In DOP, one must analyze and construct exploit manually, whereas one can write BOP exploit using SPL, an easily understandable high-level C-like language. In terms of granularity, DOP works at the gadget level whereas BOP works at the basic block level. Note that BOPC is not fully automated. BOPC's output is a set of address-value pairs for memory writes. An attacker requires to modify an address with the corresponding value from the address-value pairs to launch an attack.

The authors of DOP [51] also defined a mini-language called MINDOP to express DOP exploits, in which the attacker's payload can be specified. SPL specifies BOP exploits and allows the explicit access of virtual registers, library functions, and APIs to call OS functions. MINDOP uses a set of virtual instructions and virtual register operands. Both MINDOP and SPL are Turing-Complete languages. That means both languages support memory read/write, assignment, arithmetic operations, logical operations, control-flow transitions (*e.g.*, jump), function call, and system API calls.

It is easier to construct a BOP-based exploit than a DOP-based exploit because the gadget stitching for DOP-based exploits may depend heavily on the target program. Since the DOP gadget space is a superset of the BOP gadget space (a BOP gadget is a DOP gadget within a single basic block), it is relatively easy to find the necessary DOP gadgets from a program. But the construction of an exploit through stitching the gadgets is a challenging manual effort. Both types of exploits require memory-write primitives to stitch gadgets. However, BOPC writes memory with respect to the stack pointer and base pointer whereas we observe that the ProFTPd DOP exploit requires addresses from the address space of the ProFTPd server. The key challenge for BOP-based exploits is to find entry points. When a program's execution reaches the entry point, the BOP exploit takes over and executes the BOP payload. But, to reach the entry point, an attacker must find a vulnerability that allows the attacker to control the program control flow. Attackers usually craft external inputs with the exploit payload to trigger the vulnerability and to set up the memory for executing the exploit payload.

Although BOPC is not designed to implement a complete end-to-end attack, we provide an example to illustrate how BOPC makes the construction of a BOP exploit easier than that of a DOP exploit. To assess the difficulty of constructing a BOP exploit, we constructed a BOP exploit using the following SPL program (Listing 4) to invoke the `execve()` system call with `'/bin/sh'` as the argument. To construct the exploit, we ran the BOPC tool in Nginx (version 1.3.9) webserver. BOPC took around 30 minutes for the basic block abstraction process and around 5 minutes to

find a solution for the exploit. We ran the BOPC tool in an Ubuntu 18.04 machine with 16 GB of memory and an 8-core CPU. We ran the Nginx web server and used GDB to avoid the complication of finding and triggering a vulnerability to divert Nginx's control flow and set up an entry point for the exploit. We manually crafted the entry point of the exploit with the entry point of the Nginx program obtained through GDB. Now, when we ran the Nginx program, we observed that the invocation of the `execve()` system call and execution of the `dash` program. However, it requires non-trivial manual efforts to stitch the gadgets for constructing a DOP exploit to achieve the same attack purpose.

```
void payload()  // execve('/bin/sh') payload
{
    string prog = "/bin/sh\0";
    int argv    = {&prog, 0x0};
    __r0 = &prog;
    __r1 = &argv;
    __r2 = 0;
    execve(__r0, __r1, __r2);
}
```

Listing 4. SPL payload for invoking execve() system call.

2.7 Representative data-oriented attacks on real-world applications

The existence of single-step DDM attacks [18, 49] in programs is not new. However, the advanced data-oriented attacks are new and pose serious threats to real-world programs.

Jia *et al.* [55] utilized data-oriented attacks to bypass the same-origin policy (SOP) enforcement in the Chrome browser. By manipulating the values of in-memory flags related to SOP security policy checking (which requires an arbitrary read/write privilege), the SOP enforcement can be undermined in Chrome. Davi *et al.* [29] 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 enables malicious code injection to kernel space.

Rogowski *et al.* [88] introduced a new technique, called memory cartography, that an adversary can use at runtime to reach security-critical data in process memory, and then modify or exfiltrate the data at will. They demonstrated the feasibility of data-oriented exploits against modern browsers such as Internet Explorer and Chrome, where possible attacks range from cookie leakage to bypassing the SOP. Morton *et al.* [73] demonstrated the potential threat of data-oriented attacks against asynchronous web servers (*e.g.*, Nginx or Apache). By manipulating only a few bytes in memory, an attacker can re-configure a running asynchronous web server on the fly to degrade or disable services, steal sensitive information, and distribute arbitrary web content to clients. The attack consists of multiple steps (*i.e.*, a multi-step DDM). It starts with locating the security-critical configuration data structures of the server and exposing their low-level state by leveraging memory disclosure vulnerabilities. Then, an adversary constructs faux copies of security-critical data structures into memory by exploiting memory corruption vulnerabilities. By redirecting data pointers to faux structures, a running web server instance can be re-configured by the attacker without corrupting the control-flow integrity or configuration files on disk. However, in the end-to-end exploits, authors in [73] simulated the arbitrary write vulnerability in the recent version of Nginx, rather than exploiting a real-world vulnerability.

Table 1 summarizes these recent data-oriented attacks targeting at real-world applications. Because existing CFI-based solutions are rendered ineffective under data-oriented exploits, such threats are particularly alarming. To construct a data-oriented exploit, attackers must have an

Table 1. Recent data-oriented attacks pose serious threats against real-world programs.

Targeted Application and Year	Type	Assumption/Requirement	Capability/Attack Purpose
Chrome [55], 2016	DDM	Identified security-critical variables, and arbitrary read/write capability	Bypass the same-origin policy
Linux Page Table [29], 2017	DDM	Kernel code writable, and arbitrary read/write capability	Bypass the kernel CFI
Internet Explorer, Chrome [88], 2017	DDM	Identified security-relevant variables, and arbitrary read/write capability	Information leakage, bypass the same origin policy, etc.
Nginx [73], 2018	Multi-step DDM	Identified security-critical data structures, known unused portion of the data section, and arbitrary read/write capability	Disable or degrade services, information leakage, etc.
Nginx [37], 2015	DDM	A stack vulnerability that allows an attacker to corrupt a data pointer	Leakage of safe code pointers stored in secret locations, *i.e.*, bypassing the CPI [60] protection.
Apache HTTP Server & Glibc [93], 2014	DDM	A stack-based buffer overflow that allows attackers to corrupt data variables, data pointers, and code pointers	Derandomize code layout by learning how code is diversified without a memory disclosure vulnerability
Nginx & Sudo & Httpdx & Orzhttpd & Null Httpd & Ghttpd & Sshd & Wu-ftpd [49], 2015	DDM (Flow-Stitch)	Identified source and target data flows, and arbitrary read/write capability	Automatic construction of DDM exploits by stitching disjoint data-flows via exploiting memory errors (information leakage or privilege escalation attacks)
ProFTPd [51], 2016	DOP	Memory addresses of multiple involved data, identified gadgets/dispatchers, and arbitrary read/write capability	Private key leakage w/ ASLR
ProFTPd, Nginx, Sudo [54], 2018	DOP (BOPC)	Attack payload written in SPloit Language (SPL), and arbitrary read/write primitive	Automatic construction of BOP gadget chain or exploit payload

in-depth knowledge of the vulnerable program's exact memory layout at runtime. In comparison to the DDM attack, a DOP attack requires non-trivial engineering efforts to chain gadgets for malicious effect.

3 EXISTING DEFENSES AGAINST DATA-ORIENTED ATTACKS

We describe a three-stage model for data-oriented attacks, a taxonomy of existing defenses, and a comparative analysis of existing defenses against data-oriented attacks in this section.

3.1 Three-stage model for launching data-oriented attacks

Figure 4 illustrates the abstract view of 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 of 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.

We discuss requirements in different stages (*i.e.*, the threat model) that are essential to launching a successful DOP attack. The first three requirements apply to DDM exploits.

* 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 program's memory (*i.e.*, write capability). This assumption is reasonable since memory-unsafe languages (*e.g.*, C/C++) are still widely used today for their interoperability and speed.
* 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

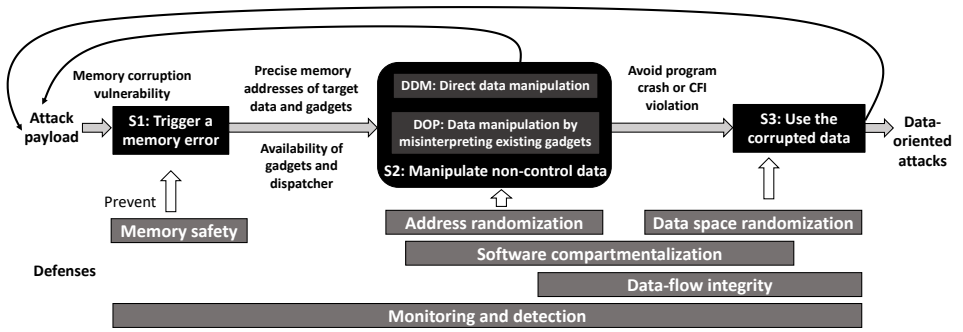

Fig. 4. Stages in data-oriented attacks and mitigation in different stages

memory disclosure vulnerabilities to circumvent the address space randomization [73]. In this case, an 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 *impact of an exploit on memory space* of the target program. For example, a continuous buffer overflow exploit may generate side effects that cause the program to crash. When launching a data-oriented exploit, attackers need to avoid any CFI violation and program crash.
* *Availability of DOP gadgets* that are reachable using a memory corruption and triggerable using an exploit
* *Stitchability of disjoint DOP gadgets.* A gadget dispatcher dispatches and executes the functional DOP gadgets. However, it is non-trivial to find gadget dispatchers in a program since they require loops with suitable gadgets and selectors controlled by a memory error.

Overall, constructing DOP exploits imposes some restrictions on the nature of the vulnerability, the context of the execution, and the defenses under deployment. Next, we discuss applicable defenses focusing on preventing these requirements from being satisfied at different points/stages. More generic memory corruption prevention mechanisms (in Stages S1 and S2) can be found in [62, 103, 105]. 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 specially focusing on data-oriented attacks have been proposed in the literature. We then discuss these defensive mechanisms in Section 3.5.

3.2 S1 Defense – Preventing the 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 overflow, use-after-free, etc. It ensures the low-level integrity of a program's data structures and avoids invalid memory accesses. Memory-safe programming languages have built-in mechanisms to protect memory errors. In contrast, *memory-unsafe* languages such as C/C++ lack built-in memory safety guarantees, hence memory errors are prevalent in programs written in these languages. They allow direct access to memory using pointers, which is a common cause of memory corruption. Nevertheless, C and C++ are still widely used programming languages today [40]. 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: approaches with low-overhead usually offer inadequate protection/coverage, while comprehensive solutions either incur a high performance-overhead or provide limited backward compatibility [99, 105]. The majority of existing memory safety solutions 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).

3.2.1 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* [74] and *HardBound* [32] 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 support was deprecated and removed from GCC and the Linux kernel [22]). 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.* [78] 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 [58] together with pointers, *e.g.*, by increasing their length [75] or by borrowing unused bits from pointers [58]. 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 primarily been deployed in clean-slate ISA designs [61], and memory-safe programming languages, *e.g.*, Cyclone [28] and Rust [109]. BIMA [61] is a hardware-assisted fat-pointer scheme for the SAFE secure computing platform [89]. 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 performance penalty, and a 3% memory overhead due to segmentation caused by alignment restrictions on BIMA pointers. *Low-fat-pointers* [34, 35] are an alternative to fat pointers compatible with commodity 64-bit hardware 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 allocation 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. Yong *et al.* [122] presented a security-enforcement tool for C programs preventing unchecked pointer dereferences. The proposed method uses static analysis to identify unsafe pointers in the program, as well as the memory locations that can be the legitimate targets of these pointers. To reduce the runtime overhead, only write instructions via unsafe pointers are instrumented to check for violations. An attack involved an attempt to write to non-allocated storage, or to an inappropriate location on the stack (*e.g.*, the return address) will be detected.

Write-Integrity Testing (WIT) [4] uses points-to analysis to compute the control-flow graph and the set of objects that can be written by each instruction in a program. Then it generates code instrumented to enforce write integrity, which prevents instructions from modifying objects that are not in the set computed by the static analysis. To achieve a low runtime overhead, WIT only instruments writes without instrumenting reads, and thus is considered an S1 defense. It does not prevent out-of-bounds reads. The authors also developed several optimizations to reduce the space and time overhead in the implementation. WIT achieves a low average overhead of 7%, and the maximum overhead is 25% across a set of CPU intensive benchmarks. It is an approximation of the spatial memory safety (*e.g.*, preventing out-of-bound writes) in terms that the data write 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. WIT mitigates use-after-free (UAF) vulnerabilities to a certain extent, because the attacker cannot use a dangling pointer to write to an object of a different equivalence class.

3.2.2 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* [26] 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. *Code-Pointer Integrity* (CPI) [60, 113] 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.* [60, 113] also introduced a relaxation of CPI with better performance properties, called code-pointer separation (CPS), to achieve better security-to-overhead trade-off. However, this solution only protects code pointers with non-control data unchecked.

Pointer Authentication (PA) [84] is a hardware pointer authenticity primitive introduced in the ARMv8.3-A processor architecture to protect programs from exploiting memory vulnerabilities. PA introduces a set of new instructions for calculating and verifying a *Pointer Authentication Code* (PAC) for pointers. The use of an unauthenticated pointer would cause a memory translation fault. Each PAC is generated using a key from a set of five different keys and a modifier. The kernel generates the five keys for each process and stores them in internal CPU registers which are not accessible from userspace code. These keys remain the same throughout the process lifetime. Out of the five keys, two are used for generating PACs for code pointers, two for data pointers, and one for general purpose uses. The modifier usually captures the contexts of pointer declarations and accesses. To store PACs, PA uses the unused bits in the virtual address of 64-bit address space. In a 64-bit Linux kernel, PA uses 24 bits for the PACs, but the size can vary based on memory scheme and address tag usages. However, PA has a few concerns regarding the PAC generation. Since PAC generation keys stay the same for the lifecycle of a process, and modifiers may have repeatability, attackers may reuse previously generated PAC and pointer pair at a later stage to replace another

PAC and pointer that uses the same modifier [65]. If the modifier does not uniquely capture the context, it might repeat in different contexts and allow such reuse attacks. For example, in return address signing using the stack pointer (SP) values as modifiers, a return address authenticated for one function can be used for another function if the SP values in both functions are the same. To address these concerns, the Pointer Authentication Run-Time Safety (PARTS [65]) technique augments the PA-based defense approach and compartmentalizes the PAC generations for different pointers in different contexts. The key idea of the PARTS approach is to utilize a pointer's type as a modifier. The type potentially captures the context in which the pointer is created and dereferenced. PARTS provides a proof-of-concept implementation based on LLVM and incurs an overhead of less than 20%.

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 could modify or dereference signed pointers and result in false alarms.

3.2.3 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) [94] is a memory error detector for Linux available in GCC and Clang/LLVM. 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 dereferences 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) [46] 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).

3.3 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, including *software compartmentalization* [36, 71, 114] and *address randomization (diversification)* [9, 107] techniques. They serve as the second line of defense, which creates a barrier for attackers trying to access target data or infer memory layout.

3.3.1 Software Compartmentalization. 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. For example, *Software Fault Isolation* (SFI) [114] 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 [36] 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* [71] 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* [118] 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 [111, 116].

3.3.2 Address Randomization. Address randomization aims to hide attack targets by randomizing the location of program segments [62], layout of the code (instruction set) [27], or layout of data [9] so that attackers are unable to predict the target data or code location from an address space. In particular, data space randomization [8, 9, 44] aims to randomize the locations of data stored in program memory at runtime to make the locations unpredictable, and thus reducing the possibility that attackers can leak security-critical memory addresses or manipulate the content of targeted data. ASLR [107], also known as coarse-grained ASLR, randomly relocates only the base addresses of the stack, heap, code segments, and shared libraries on each execution of a program. However, the internal layout of a segment or module remains unchanged. Also, to relocate the code and data segments of the main executable of a program, it is necessary to run the program as position-independent code (*i.e.*, with the PIE feature enabled). Fine-grained ASLR techniques relocate the internal layout of a segment or module up to different granularities such as function-level [23, 44, 56], basic block-level [19, 57, 115], instruction-level [47], and machine register-level [27, 48]. Attackers can still figure out the fine-grained address space layout of a program within a few seconds [3] using advanced attack techniques [101].

Though strong randomization can stop memory corruption attacks with a high probability, the protection is confined to all data/addresses that are randomized/encrypted. In practice, to avoid a significant performance degradation, not all data/addresses are protected by randomization defenses [105]. On the other hand, information leaks can undermine randomization techniques [101]. ASLR-Guard [69] can add an extra layer of protection for randomization techniques suffering from information leaks. Besides, ASLR-Guard [69] can render the code pointer leak through a data pointer useless by separating the data and code using a custom linker that generates new relocation information to fix the offset between the data and code. In addition, data/address encryption based solutions are not binary compatible (*i.e.*, protected binaries are incompatible with unmodified libraries) [105].

3.4 S3 Defense – Preventing/detecting use of corrupted data

Data-Flow Integrity (DFI) [15] mitigates data corruption before the manipulation takes effect. Before each read instruction, DFI ensures that a variable can only be written by a legitimate write instruction which can be derived by reaching definitions analysis. For each read instruction for reading a value, it statically computes the set of write instructions that may write the value, and assigns an identifier to each definition. DFI enforces a simple safety property, *i.e.*, whenever a value is read (used), the definition identifier of the instruction that wrote the value should be in the set of reaching definitions for the read. DFI is different from the data write integrity checking in S1 defense (such as WIT [4]) in that it enforces data integrity for memory reads. Thus, the

enforcement of DFI happens in Stage 3. WIT prevents data manipulation by protecting against invalid/unintentional memory writes, but with reads left unchecked. In addition, since DFI could potentially limit the DOP gadget availability, it is active at both S2 and S3 in our three-stage model.

DFI enforcement can prevent both control-data (*e.g.*, overwriting the return address) and non-control-data attacks. 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 [51] due to the frequent read instruction checking. Intra-procedural DFI incurs 44% and inter-procedural DFI incurs 103% runtime performance overhead, respectively, and approximately 50% space overhead for instrumentation [15]. Hardware-based DFI, *e.g.*, HDFI [102], is efficient, but limited by the number of simultaneous protection domains it can support. Carlini *et al.* [14] 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 prevent the use of corrupted data. For example, when a corrupted pointer is referencing memory in another protection domain, it thwarts the dereference operation. Besides, some techniques [30, 106] can prevent the execution of code pointed by corrupted data or memory. Heisenbyte [106] is such a technique that utilizes "destructive code read". The "destructive code read" allows the execution of code as part of the normal flow of a program and restricts the execution of the same code when used in a dynamic code-reuse attack. Another technique called Isomeron [30] tackles the usage of existing code pointed by corrupted data or memory by randomizing the execution paths.

Data Space Randomization (DSR) [9] encrypts data (*i.e.*, randomizes the representation of data objects with a unique random mask) stored in memory, rather than randomizing the location of data objects. When a variable is used, its value will be first derandomized by using the unique random mask associated with the variable. By using different masks for different variables, DSR ensures that even if the attacker manages to overwrite a target variable, the attacker is only able to write a random value into it rather than the intended value. Instead of using a different mask for each data object, Data Randomization [13] groups data objects into different classes, assigns a random mask for each class, and generates instruments code to XOR data read from or written to memory with the corresponding mask. As a result, accesses that violate the read or write integrity have unpredictable results. Another recent DSR technique called CoDaRR [85] continuously rerandomizes the masks used in variables with load and store instructions while being transparent to program execution. The dynamic nature of CoDaRR prevents disclosure attacks. DSR [9, 85] and Data Randomization [13] are effective against non-control data attacks by preventing attackers from using the corrupted data in the memory space (although they can overwrite target data variables). And thus they actually take effects at Stage 3. Nevertheless, masking/unmasking every memory access inevitably incurs nontrivial runtime overheads, which hinders their practical deployments.

Szekeres *et al.* [105] highlights in their paper that real-world software exploits are still possible because memory vulnerabilities continue to grow and currently deployed defenses are being bypassed. Thus, program anomaly detection [21, 97, 120, 124] may complement the aforementioned mitigation techniques, and may serve as the last line of defense against data-oriented attacks. As shown in Figure 4, passive monitoring based program anomaly detection has the potential to detect anomalous program behaviors exhibited in all the three stages.

3.5 Defense mechanisms specially against data-oriented attacks

Besides the generic memory corruption prevention mechanisms which can be applied to defeat data-oriented attacks, here we discuss detection and prevention techniques explicitly focusing on data-oriented attacks.

YARRA [90] is a C language extension that validates a pointer's type for *critical data types* annotated by developers. It 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 the whole program protection, it incurs performance overhead around 400%–600%. Besides, YARRA relies on the programmers' manual annotations, which is undesirable for complicated programs.

HardScope [77] is a hardware-assisted variable scope enforcement approach to mitigate data-oriented attacks. It performs intra-program memory isolation based on C language variable visibility rules derived during program compilation. On each memory access (*i.e.*, load/store), HardScope enforces that the memory address requested is in the accessible memory areas. Nyman *et al.* [77] demonstrated the effectiveness of HardScope for the RISC-V architecture, by introducing a set of seven new instructions. HardScope instruments instructions at compile-time and enforces memory access constraints at runtime. HardScope's performance overhead is reasonable with 3.2% in embedded benchmarks. Although HardScope significantly reduces the usefulness of DOP gadgets and thwarts Hu *et al.* [51]'s example attacks, HardScope cannot guarantee the absence of DOP gadgets in arbitrary programs.

PrivWatcher [17] is a framework for monitoring and protecting the integrity of process credentials (*i.e.*, *task_struct* that describes the privileges of a process in the Linux kernel) against non-control data attacks. PrivWatcher provides non-bypassable integrity assurances by relocating process credentials into a safe region, code instrumentation and runtime data integrity verification. It incurs more than 94% overhead for applications that involve installing new *task_struct* structures to processes.

Hardware-Assisted Data-flow Isolation (HDFI) [102] extends the RISC-V architecture to provide an instruction-level isolation by tagging each machine word in memory (also known as the tag-based memory protection). The one-bit tag of a memory unit in HDFI is defined by the last instruction that writes to this memory location. At each memory read instruction, HDFI checks if the tag matches the expected value. However, unlike software-enforced DFI, HDFI only supports two simultaneous protection domains.

PT-Rand [29] aims to protect a data-oriented attack against kernel page tables to bypass CFI-based kernel hardening techniques. To mitigate the manipulation of page tables, PT-Rand randomizes the location of the page tables. PT-Rand incurs a low overhead of 0.22% for common benchmarks on Debian. However, it is still possible attackers undermine these schemes if the secret information (*e.g.*, randomization secret) is leaked or inferred [77].

Both C-FLAT [2] and eFSA [20] identify non-control-data attacks through their effects on the control-flow of a program in embedded systems, such as Internet of Things (IoT) or Cyber-Physical Systems (CPS). C-FLAT [2] enables remote attestation of an application's control-flow path, which allows a verifier to detect control-flow deviations launched via code injection, code-reuse, and certain non-control-data attacks. Since C-FLAT computes an aggregated authenticator of the program's control flow, including branches and function returns, it can detect non-control-data attacks that affect a program's sequence of executed instructions or the number of loop iterations. eFSA [20] is an event-aware finite-state automaton model for detecting non-control-data attacks against programs in CPS. It takes advantage of the event-driven nature of CPS control programs and

incorporates event checking in anomaly detection. It detects non-control-data attacks if a specific physical event is missing along with the corresponding event dependent executed instructions.

CVI (Critical Variable Integrity) [104] verifies define-use consistency of critical variables for embedded devices. The define-use consistency enforces that 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 [15], CVI is based on compile-time instrumentation and frequent runtime checking, which incurs a high overhead for the complete protection.

Hardware-based detector I. This detector [110] utilizes Hardware Performance Counters (HPCs) to collect hardware events related to instructions retired, cache-misses suffered, and branches miss-predicted for detecting data-only exploits. The authors collect 12 hardware events during the normal and abnormal executions of OpenSSL and train a multi-class support vector machine model to classify normal and abnormal behaviors. Two fundamental limitations of this HPC-based detector is that *i*) the co-executing programs significantly affect HPC-based events and *ii*) HPC-based events are dependent on instruction set architectures (ISAs).

Hardware-based detector II. This detector [66] utilizes the HPC events as short time series by monitoring various execution regions of a vulnerable program. Especially, the samples include the region before, during, and after executing a vulnerable region of a program. This time series approach enables more fine-grained detection than using only hardware events. Classification algorithms such as the Stacked Denoising Autoencoder and Echo State Network classifiers are around 98% accurate for detecting data-oriented exploits.

In summary, HDFI [102], PrivWatcher [17], PT-Rand [29], and CVI [104] protect specifically non-control data. HardScope [77] can protect against all DOP attacks that violate variable visibility rules at runtime. However, its main drawback is that the new hardware extensions [102] set a high bar for deployment. On the other hand, the two general approaches DFI [15] and YARRA [90] incur a high performance-overhead at runtime. In general, the HPC-based hardware events are significantly affected by co-existing programs. Thus, the filtration of the hardware events that are produced by co-existing programs is a critical step for obtaining accuracy and reliability by HPC-based detectors. The time-series approach used by the hardware-based detector II [66] makes the detector less independent of the underlying applications. Different from the above works focusing on traditional programs, C-FLAT [2] and eFSA [20] target at detecting non-control-data attacks in embedded systems by monitoring their side-effects on control-flow behavior.

4 CHARACTERIZATION OF CONTROL-FLOW ANOMALIES IN DATA-ORIENTED ATTACKS BASED ON HARDWARE-ASSISTED CONTROL-FLOW TRACING

In this section, we conduct a branch correlation analysis and frequency-based analysis to characterize control-flow anomalies during the course of data-oriented attacks based on hardware-assisted control-flow tracing. We consider the scenario where one can efficiently trace control flows, *i.e.*, recording indirect control transfers and conditional branches. With new hardware development, namely Intel Processor Trace (PT)[2], efficient real-time control-flow monitoring has the potential to be widely deployed. There exist cases where data-oriented attacks could be detected by correlating or aggregating multiple control-flow observations. In the branch correlation analysis approach,

[2]PT is a low-overhead hardware feature on Intel CPUs that enables the construction of the complete control flows during program execution [64].

we show that programs under data-oriented attacks may exhibit incompatible branch behavior. In the frequency anomaly-based approach, we show that the frequency distributions of a program's normal executions might be different from the program's execution under data-oriented attacks. It is important to note here that our analysis in this section is an initial characterization effort to understand how DOP attack-behaviors differ from normal executions. The characterization insights could be starting points for building potential defenses in the future. Our intention is to stimulate a discussion about potential ways (in addition to the existing defense mechanisms described in Section 3) to defend against data-oriented attacks.

4.1 Control-Flow Characterization in Data-Oriented Attacks

The *necessary condition* for a data-oriented attack to exhibit control-flow anomalies is that the attack directly or indirectly affects the flow of a program's execution. Typically, uses of non-control data in a program can be classified either as predicate uses or non-predicate uses.A predicate-use directly affects the control flow. While a non-predicate use may affect the computation or the output of a program [86]. We categorize the cases where data-oriented attacks are impossible to detect (*i.e.*, undetectable cases) and cases where such attacks may be detected (*i.e.*, detectable cases) by control-flow tracing. For the detectable cases, we provide experimental observations in Section 4.3 and Section 4.4.

Undetectable cases. Detecting data-oriented attacks using control-flow tracing requires that an attack manifests incompatible/unusual control-flow behaviors, *e.g.*, incompatible branch behaviors or frequency anomalies. However, 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 transfers, such an attack is undetectable by PT-based control-flow tracing. We list typical undetectable cases (after the phase of triggering memory errors), which are mainly direct data manipulation (DDM) attacks.

* Corrupting user identity for privilege escalation: Simply corrupting user identity 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 typically performs malicious actions after obtaining the new privilege, *e.g.*, launching a shell. Such malicious actions can be easily detected by control-flow tracing.
* Corrupting configuration data: Corrupting configuration data via format string vulnerabilities may evade PT-based detection, since format string vulnerabilities allow a single memory write without a side effect on control-flow behavior.
* Constructing exfiltration channels for information leakage: Attackers exploit an existing information outlet (also known as sink functions such as `printf` or `send`) 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.

Detectable cases. We observe that data-oriented attacks can potentially lead to three types of anomalous control-flow behaviors.

* *Incompatible branch behavior*: Manipulating a predicate-use variable (*e.g.*, decision-making 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, it is likely the data manipulation incurs incompatible branch behaviors that can be detected by control-flow tracing. For example, in Listing 1, the conditional branches in lines 5 and 16 are correlated, since they both are data-dependent on the variable `authenticated`. Suppose `authenticated` is corrupted at line 6 and there is no write to `authenticated` after the data manipulation. As a result, `while(!authenticated)` in line 5 returns `true`, but `if(authenticated)`

in line 16 also returns `true`. We observe an incompatible branch behavior, which is detected when the corrupted variable `authenticated` is used in line 16 (*i.e.*, in Stage 3 of the attack). In addition, a continuous buffer overflow may generate side effects on control-flow behavior, which could result in an incompatible control-flow path observable in Stage 2. 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 [108], and found that 5 out of 14 overflows have an impact on decision-making variables involved in predicate expressions. In Section 4.3, we experimentally characterize the branch correlation behaviors of ProFTPd under the DOP attack.

* *Macro-level interaction frequency anomaly*: In DOP (also BOP) attacks, an attacker normally needs to interact with a vulnerable program to repeatedly corrupt variables to achieve the attack purpose and avoid segmentation faults. This attack activity inevitably results in frequency anomalies during the client-server interaction, which can also be captured by control-flow tracing. For example, in the ProFTPd DOP attack introduced in Section 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.
* *Micro-level control-flow frequency anomaly*: Short control-flow paths may exhibit unusual execution frequencies. For instance, corrupting variables which directly or indirectly control loop iterations can 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 [14] and resource wastage attacks [7] may also lead to unusual control-flow frequencies. In Section 4.4, we experimentally compare the frequency differences of both the macro-level interactions and micro-level control-flows in normal ProFTPd executions and under a DOP attack.

4.2 Tracing Indirect and Conditional Branches with PT

In this section, we briefly explain the control-flow information recorded by PT, which is necessary to understand our experimental results. To capture control-flow information, PT records target addresses of indirect branches (*i.e.*, TIP packets/events for the indirect `call`, indirect `jmp`, and `ret`) and taken/non-taken decisions of conditional direct branches (*i.e.*, TNT packets/events). PT uses a highly compressed data format to log traces at runtime for efficiency. For example, it uses one bit to indicate taken or not-taken for a conditional branch.

Recent research has shown PT's applicability for on-line security enforcement to defend against control-flow attacks (referred to as dynamic CFI solutions). Griffin [42] is an operating system mechanism (running in the kernel) that leverages the PT feature to enforce CFI policies. PT-CFI [45] and FlowGuard [67] are two backward-edge control-flow violation detection systems using PT tracing. To address the over-approximation problem of control targets in forward-edge CFI, PittyPat [33] utilizes PT to track basic block execution to compute the legal control transfer target sets through runtime path-sensitive point-to analysis. But PittyPat still makes approximations, as an incomplete execution context is used in its points-to analysis. μCFI [50] improves PittyPat by recording full execution context with PT to perform an accurate points-to analysis, and thus getting a unique code target for each indirect control-flow transfer. However, none of these existing works investigates the PT-based detection against data-oriented attacks. In what follows, we look into the possibility of PT-based detection against DOP attacks, which can also be applied to detect DDM attacks.

4.3 Characterization of Incompatible Branch Behavior

In branch correlation analysis, we utilize the correlations among control-flow branches to detect incompatible branch behavior. We define two types of branch correlations: *i)* subsume and *ii)*

mutually exclusive. We say a branch condition/predicate BR_i subsumes another branch condition/predicate BR_j when BR_i's true branch satisfies the branch BR_j to take the true branch. For example, $x > 10$ subsumes $x > 5$. BR_i and BR_j are mutually exclusive if they always take different branches. For example, $x > 10$ and $x < 5$ are mutually exclusive. Incompatibility in runtime subsume and mutually correlations may indicate unusual program behaviors.

4.3.1 How to identify branch correlation. We can use the satisfiability modulo theories (SMT) solver Z3 [31] to determine subsume or mutually exclusive correlation between branches. To determine the subsume correlation between two branches, *e.g.*, (BR_1) and (BR_2), we feed the branch predicates as constraints to the SMT solver. However, we feed the logical negation of one of the branch conditions, *i.e.*, (BR_1) and *Not*(BR_2). If the SMT solver can not find a solution for the combined constraints, *i.e.*, (BR_1) and *Not*(BR_2), this indicates that BR_1 subsumes BR_2, otherwise they are mutually exclusive.

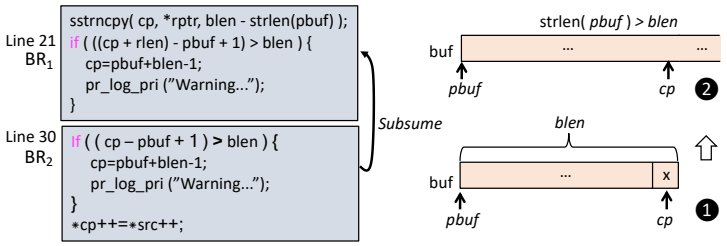

Fig. 5. Branch correlation and an incompatible branch behavior in the ProFTPd DOP attack

We utilize the above technique to identify two branches from ProFTPd's vulnerable function *sreplace* in Listing 3. The conditional branches in lines 21 and 30 of Listing 3 are correlated. Both are data dependent on the same variables *cp*, *pbuf*, and *blen*. Figure 5 shows these two correlated branches, denoted as BR_1 and BR_2, respectively. To derive the subsume relationship from BR_2 to BR_1, we first add the predicate BR_2 as a constraint into Z3 solver. However, *cp* gets redefined in line 31 (Listing 3), which is true-control-dependent on the branch in line 30. Thus, we need to replace BR_2 with 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. Because the variable *rlen* is the length of a non-null string (derived from the source code), we add *rlen > 0* as an additional constraint. At the end, the solver returns *UNSAT* (unsatisfiable), and thus we derive that BR_2 subsumes BR_1, *i.e.*, if BR_2 returns *true*, BR_1 should also take the *true* branch.

4.3.2 Existence of runtime incompatible branch behavior in ProFTPd DOP. To examine the existence of any runtime incompatible branch behavior in the ProFTPd DOP attack, we observe and analyze the execution patterns of the attack on a 32-bit Ubuntu 16.04 computer with Intel Processor Trace (PT) enabled. To automate the observation and analysis process, we modify the original ProFTPd DOP attack. The modified script and traces are available at https://github.com/doppt/data-oriented-attacks. Now, during the course of the ProFTPd DOP attack, the attacker first triggers the memory corruption error in *sreplace*, by filling up *buf* (where BR_2 returns *true*) and overwriting *buf*'s terminator with a non-zero byte (in line 33 of Listing 3), as shown in ❶ in Figure 5. Since *buf*'s last character is a non-zero value, it becomes a non-terminated string. As a result, *strlen(pbuf) > blen* (❷ in Figure 5), which enables the attacker to corrupt the local variables such as *cp* and *blen* in line 20. To bypass the overflow checking in lines 21-27 in the following iterations, the attacker needs to make sure that the predicate in line 21 returns *false*. From the PT trace, we observe the predicate in line 21 (*i.e.*, BR_1) takes the *false* branch. Since BR_2 subsumes BR_1, and BR_2 has taken the *true* branch, the runtime branch behavior of BR_1 and BR_2 are incompatible. Note that the observed incompatible branch behavior is not specific to DOP attacks. For any DDM attack against the

ProFTPd, as long as the attacker exploits the same vulnerability (CVE-2006-5815 [76]), we should observe this incompatible behavior.

4.3.3 Challenges of identifying branch correlations. Conditional branches are optional execution paths. Prior studies show that conditional branches are not completely independent of each other [126]. 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) [72] [39]. Zhang *et al.* [126] 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 [126].

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.

4.3.4 Potential of branch correlation. Our LLVM-based [68] 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.

Our case study of the ProFTPd DOP attack demonstrates that correlated branches are useful for identifying incompatible branch behaviors. We want to point out that identify deterministic relationships among correlated branches with arbitrary predicate expressions is a challenging and unsolved problem. Here, we only identify coarse-grained correlated conditional branches that have either direct or indirect joint data-dependency. Our purpose is mainly to characterize the potential prevalence of branch correlation in benchmark programs, and motivate future research on this interesting topic. As an assessment, we performed our branch correlation analysis on eight programs. They include four Linux utility programs (*flex*, *grep*, *gzip*, *sed*) from the Software-artifact Infrastructure Repository (SIR) [100] and four vulnerable programs (*wu-ftpd*, *orzhttpd*, *ghttpd*, *sudo*) from the FlowStitch benchmarks [49].

To capture coarse-grained branch correlations with arbitrary predicate expressions, we categorize the correlated and data-dependent branches into three categories, including *i) direct* data-dependent branches that share at least one common predicate variable; *ii) indirect* data-dependent branches that use different predicate variables, but they are data-dependent on at least one common variable; and *iii) simple* branches, where a branch predicate simply compares a variable with a constant value, *e.g.*, the conditional branches in lines 5 and 16 in Listing 1.

We developed a coarse-grained branch correlation analysis tool based on LLVM [68], which identifies conditional branches that have joint data-dependency. Our tool handles *inter-procedural* branch correlations and *arbitrary predicate expressions*. Table 2 shows the results of our coarse-grained branch correlation analysis. Overall, 24% of the branches exhibit simple forms of conditional predicates, and 18% of the branches with simple forms are correlated (including direct and indirect correlations). For the direct correlations with simple forms (*i.e.*, correlated branches use the same predicate variable), we used the SMT logic solver Z3 [31] to determine any "subsume" or "mutually exclusive" relationship. Our results show that for a limited number of branch correlations (around 2%), we can directly derive the deterministic correlation relationship (denoted as "directly derivable simple BRs" in Table 2).

Table 2. Coarse-grained branch correlation analysis in benchmarks

Application	Total BRs	Correlated BRs	Simple BRs	Correlated Simple BRs	Directly Derivable Simple BRs
flex	1142	813 (71%)	557 (49%)	356 (31%)	62 (5%)
grep	1664	1456 (87%)	278 (17%)	216 (13%)	32 (2%)
gzip	737	533 (72%)	241 (9%)	169 (23%)	60 (8%)
sed	1081	1017 (94%)	172 (16%)	142 (13%)	12 (1%)
wu-ftpd	2781	1943 (70%)	688 (25%)	398 (14%)	84 (3%)
orzhttpd	35	23 (66%)	13 (37%)	7 (20%)	0 (0%)
sudo	675	499 (74%)	163 (24%)	116 (17%)	0 (0%)
ghttpd	107	90 (84%)	16 (15%)	11 (10%)	0 (0%)
Average	1028	77%	24%	18%	2%

We observed 77% of the branches have at least one correlated branch, *i.e.*, given BR_i, we can find at least one branch $BR_j \neq BR_i$, where BR_i and BR_j have joint data dependency. This result suggests the prevalence of branch correlations with complex predicate expressions in a program, which can be potentially used as checkpoints to detect incompatible branch behaviors (*e.g.*, first using symbolic execution or dynamic analysis techniques to identify deterministic relationships among these correlated branches). Note that our analysis only captures the *coarse-grained* branch correlations as opposed to the deterministic relationships among branches. However, it is nontrivial to derive the deterministic relationship between inter-procedural correlated branches with complex predicate expressions through static analysis. Accurate determining inter-procedural branch correlation can be complicated, which involves comprehensive point-to analysis, value-flow analysis, and data-flow analysis. Identifying deterministic relationships among correlated branches with arbitrary predicate expressions is an interesting research problem, but it is beyond the scope of this work and thus we leave it as future work.

4.4 Characterization of Frequency Anomalies

In frequency-based analysis, we discuss the impact of DOP attacks on two types of frequency distribution, *i)* macro-level interaction frequencies and *ii)* micro-level control-flow frequencies.

4.4.1 Macro-level interaction frequencies. To observe interaction frequency anomalies under the DOP attack, we derived the FTP commands sent from clients by tracing control-flow transfers of the FTP command dispatcher function *_dispatch* in the ProFTPd server program. PT captures the control-flow transfers from *_dispatch* to different command handlers, *e.g.*, *core_cwd* indicates that the command *CWD* (*i.e.*, change working directory) has been received. For frequency distributions of

normal operations (*i.e.*, the baseline FTP interaction frequencies), we used the LBNL-FTP-PKT [63] dataset. It contains all incoming anonymous FTP connections to public FTP servers at the Lawrence Berkeley National Laboratory over a ten-day period, a total of 21,482 FTP connections. Each connection session is considered as a behavior instance, and we extract FTP commands in each connection from the dataset.

We computed the frequency distributions of *2*-gram FTP command sequences. Each 2-gram transition corresponds to a high-level execution feature. We applied the Principal Component Analysis (PCA) technique for dimension reduction, as such a distribution-based profiling produces a large number of features. We adopted the X-means clustering approach [81] to cluster all behavior instances in baseline FTP command sequences, where the center of each of the X-clusters represents a normal program execution context.

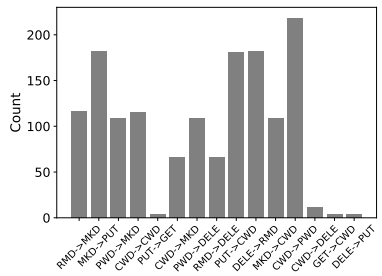

Fig. 6. For macro-level interaction frequencies, 2-gram distribution of FTP commands within a connection during the ProFTPd DOP attack.

Fig. 6 shows the macro-level frequency distribution of 2-gram FTP commands within a connection during the ProFTPd DOP attack. Over the course of the attack, it involves more than 1,000 client-server FTP commands. In contrast, the average interactions per session in the normal LBNL-FTP-PKT dataset [63] is 41. Fig. 7 illustrates the X-clustering for 2-grams of FTP commands with PCA reduction to 3-dimension. The DOP instance (*i.e.*, red triangle) does not belong to any normal clusters (*i.e.*, blue dots). These results suggest that the client-server interactions under the DOP attack drastically differ from the baseline executions.

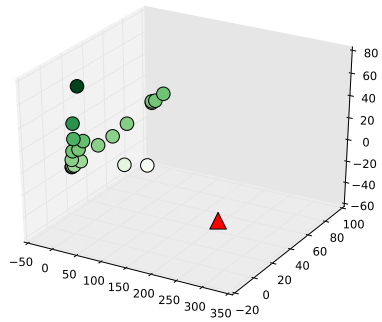

Fig. 7. For macro-level interaction frequencies, X-clustering for 2-grams of FTP commands with PCA reduction to 3-dimension using LBNL-FTP-PKT dataset [63]. The ProFTPD-based DOP attack involves an abnormally high number of client-server interactions.

4.4.2 Micro-level control-flow frequencies. Short control-flow paths may exhibit unusual execution frequencies in data-oriented attacks. For instance, corrupting variables which directly or indirectly control loop iterations can cause such frequency anomalies.

As mentioned in Section 2.4, changing the working directory (*i.e.*, `CWD` command) triggers the invocation of vulnerable *sreplace* function. In ProFTPd DOP attack, an attacker needs to craft

.message files (*i.e.*, as malicious payloads) to repeatedly fill up the allocated buffer *buf* and write bytes beyond the buffer in *sreplace*, which exhibits anomalous behaviors of control-flow transfers (*i.e.*, significant frequency anomalies of control-flow transfers in *sreplace*). We defined all control-flow transfers in each *sreplace* invocation as a behavior instance, following the approach in [96]. Since it is difficult to harvest *.message* files from old version FTP servers, in this experiment, we randomly generated 1000 *.message* files without triggering the overflow as the baseline executions.

Given a new behavior instance (*i.e.*, PT traces) observed in the detection phase, we measured the distance between the new observation and each of cluster centers in the training dataset. If it does not belong to any existing cluster (*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 *sreplace* function.

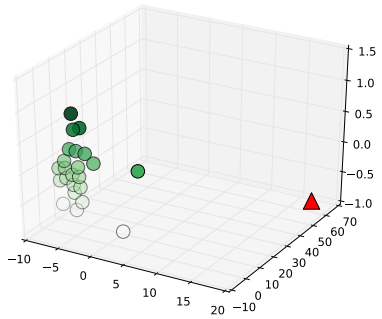

Fig. 8. For micro-level control-flow frequencies, X-clustering for 2-gram control-transfers with PCA reduction to 3-dimension in *sreplace*. The ProFTPD-based DOP attack exhibits a unique pattern of control-flow transfers compared to baseline executions.

The feature extraction and dimension reduction procedures are similar to the macro-level analysis described above. After applying PCA, we reduced the original high-dimensional data to 3-dimensional data and then performed the X-clustering. Our result comparing the control-flow frequency properties in *sreplace* is shown in Figure 8. The baseline dataset is clustered into 23 clusters. Similar to Figure 7, the DOP instance is an obvious outlier. BOP is likely to have a similar type of frequency anomalies because BOPC typically utilizes multiple arbitrary memory read/write primitives to modify the memory state of the target binary.

4.4.3 Limitation of control-flow frequency analysis. Though DOP attacks are expressive and potentially powerful, they may manifest unusual control-flow behavior in different dimensions and stages. In the above case study, we demonstrated the possibility 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. Note that incompatible control-flow paths may also result in frequency anomalies. Therefore, statistical modeling of control-transfers has the potential to detect not only anomalous frequency of legitimate control-flow paths, but also incompatible control-flow path anomalies in data-oriented attacks. It is not a DOP-specific and new defense, but could be potentially applied to defend against data-oriented attacks [98]. However, as many anomaly detection approaches [121], frequency analysis based detection is vulnerable to defense-aware adaptive attacks, where an attacker may obscure control-flow frequency anomalies by intentionally interleaving benign control-flows or slowing down the attack to evade detection. Therefore, we consider control-flow frequency analysis as a complementary mitigation technique to defeat data-oriented attacks.

5 CONCLUSION AND FUTURE RESEARCH OPPORTUNITIES

In this work, we systematized the current knowledge of data-oriented exploits and applicable defense mechanisms. We experimentally explored the possibility of using low overhead tracing techniques, namely PT, for characterizing data-oriented attacks. We hope that this systematization will stimulate a broader discussion about possible ways to defend against data-oriented attacks. We highlight some interesting future directions in this area.

Automation of Small Footprint DOP Attacks. An interesting research direction is how to minimize the footprints (*i.e.*, side effects) of a DOP attack while achieving the same attack goal. Our experiments in Section 4 showed that the DOP attack clearly exhibits some anomalies in the control-flow behavior. It alters the correlation or statistical properties of control flows. DOP/BOP gadgets may have different impacts on control-flow behaviors. Attackers may prefer data-oriented gadgets that cause a minimum deviation from normal executions. Such a selection process requires automation to be efficient. Besides automation, one also needs to define metrics to measure the footprints, *i.e.*, the amount of alteration caused by a DOP execution. Ispoglou *et al.* [54] made the first step towards automating data-oriented programming through a powerful Block Oriented Programming Compiler (BOPC). Searching for gadget chains under specific constraints is an interesting research direction.

Assessment of Programs' Susceptibility to Data-Oriented Attacks. Such a characterization – statically or dynamically – would help one understand the threats that CFI cannot protect against. A promising direction is to quantify the degree of control-flow decisions that are dependent on adversarially controlled data (*e.g.*, user input). Such a characterization also helps prioritize the defense effort, enabling one to address programs with the highest susceptibility first.

Low False Positive PT-based Anomaly Detection. DOP attacks exhibit occasional anomalous execution behaviors at runtime, as we have demonstrated in Section 4. However, to design a successful anomaly detection solution targeting DOP, much more work is needed. Specifically, one needs to show the instruction-level detection does not trigger many false positives in normal executions. Virtually all existing learning-based program anomaly detection demonstrations are at the higher system-call and method-call levels. Reasoning instruction-level PT traces for anomaly detection is challenging.

Deep Learning for Control-Flow Behavior Modeling. Non-control data violations may have impacts on control-flows in different locations with long distances in a program. How to detect incompatible control-flow paths, given a relatively long control-flow sequence, is challenging. Deep learning techniques have shown promises in detecting anomalies in different applications. An interesting research direction is to apply deep learning algorithms to model program behaviors for anomaly detection. For example, Recurrent Neural Network (RNNs), especially Long Short-Term Memory (LSTM) models, can be leveraged to capture temporal relation contained in univariate or even multivariate data. Such techniques may have the potential to detect incompatible control-flow paths given an extreme long control-flow sequence [70]. However, one challenge of the deep learning based detection is the lack of labeled attack data given the difficulty to construct different DOP/BOP exploits. In addition, attackers may exploit adversarial machine learning techniques to evade detection by obfuscating control-flow behaviors under data-oriented attacks.

Anti-specification Database for Detecting Data-Oriented Attacks. Anti-specifications–a specification-based technique that describes the violation of legal data flows–can aid the detection of data-oriented attacks. Specifications are what a program should do, whereas anti-specifications [112] are what a program should not do. Anti-specifications aim to capture events that may be part of an exploit. For example, attackers trigger most exploits using unguarded external inputs. Anti-specifications provide a characterization of the impact of a program's use of unguarded external input on the

downstream data-flow of the program. The impact analysis of a program's external input is a key step for preventing the gateways of data-oriented exploits. Besides, anti-specifications can also capture input-dependent predicates used in loops or conditions. These controllable predicates can alter program behavior. Other types of anti-specifications may capture data leaks [11] or data pointer corruption. Thus, anti-specifications can help improve the detection of data-oriented exploits by identifying the gateways and generic components used in those exploits.

One key benefit of anti-specifications is the construction of an anti-specification database. This database is a collection of anti-specifications in a specific format, which helps screen programs against the anti-specifications stored in the database. It could also be used to generate new anti-specifications by merging two or more anti-specifications. The key difference of this anti-specification-based approach from a signature-based approach is that anti-specifications are not specific to an exploit. The approach attempts to produce anti-specification by breaking an exploit into its modular events or components and extracting anti-specifications from them. Even though the same or similar exploits vary from one program to another, we may observe many common components (*e.g.*, exploit entry, gadget lookup, gadget stitching, and triggering) when unfolding the exploits. As a result, anti-specifications built from such common components can be more generally applied across multiple programs.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

[2] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. 2016. C-FLAT: Control-FLow ATtestation for Embedded Systems Software. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[3] Salman Ahmed, Ya Xiao, Kevin Z Snow, Gang Tan, Fabian Monrose, and Danfeng Yao. 2020. Methodologies for quantifying (Re-) randomization security and timing under JIT-ROP. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1803–1820.

[4] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing memory error exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 263–277.

[5] Starr Andersen and Vincent Abella. 2004. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies.

[6] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1342–1353.

[7] Arati Baliga, Pandurang Kamat, and Liviu Iftode. 2007. Lurking in the shadows: Identifying systemic threats to kernel data. In *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 246–251.

[8] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M. Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, et al. 2018. Hardware Assisted Randomization of Data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 337–358.

[9] Sandeep Bhatkar and R. Sekar. 2008. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 1–22.

[10] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 268–279.

[11] The Heartbleed Bug. 2020. http://heartbleed.com. Accessed April 03, 2020.

[12] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 1–33.

[13] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. 2008. *Data randomization*. Technical Report. Technical Report TR-2008-120, Microsoft Research, 2008.

[14] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*. 161–176.

[15] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. 147–160.

[16] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 380–394.

[17] Quan Chen, Ahmed M Azab, Guruprasad Ganesh, and Peng Ning. 2017. Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 167–178.

[18] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, Vol. 5.

[19] Yue Chen, Zhi Wang, David Whalley, and Long Lu. 2016. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY '16)*. ACM, 50–61.

[20] Long Cheng, Ke Tian, and Danfeng Yao. 2017. Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks. In *Annual Computer Security Applications Conference (ACSAC)*.

[21] Long Cheng, Ke Tian, Daphne Yao, Lui Sha, and Raheem A. Beyah. 2019. Checking is believing: event-aware program anomaly detection in cyber-physical systems. *IEEE Transactions on Dependable and Secure Computing* (2019).

[22] Matthew Cole and Aravind Prakash. 2020. Simplex: Repurposing Intel Memory Protection Extensions for Information Hiding. arXiv:2009.06490 [cs.CR]

[23] Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi. 2016. Selfrando: Securing the tor browser against de-anonymization exploits. *Proceedings on Privacy Enhancing Technologies* 2016, 4 (2016), 454–469.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[25] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX Security Symposium*, Vol. 98. San Antonio, TX, 63–78.

[26] Stanley Crispin Cowan, Seth Richard Arnold, Steven Michael Beattie, and Perry Michael Wagle. 2010. PointGuard: method and system for protecting programs against pointer corruption attacks. US Patent 7,752,459.

[27] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 763–780.

[28] Cyclone. 2002. http://cyclone.thelanguage.org/. [Accessed 08-12-2019].

[29] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *NDSS*.

[30] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. 2015. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming.. In *NDSS*.

[31] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, 337–340.

[32] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 103–114.

[33] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *USENIX Conference on Security Symposium*. 131–148.

[34] Gregory J Duck and Roland HC Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*. 132–142.

[35] Gregory J Duck, Roland HC Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers.. In *NDSS*.

[36] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. 75–88.

[37] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the point(er): On the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 781–796.

[38] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 901–913.

[39] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. 1998. An analysis of correlation and predictability: what makes two-level branch predictors work. In *Proceedings. 25th Annual International Symposium on Computer Architecture*. 52–61.

[40] TIOBE Index for November. 2020. https://www.tiobe.com/tiobe-index/. Accessed November 30, 2020.

[41] Aurélien Francillon and Claude Castelluccia. 2008. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and Communications Security*. 15–26.

[42] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 585–598.

[43] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1009–1022.

[44] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium (USENIX Security 12)*. 475–490.

[45] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*. 173–184.

[46] Hardware-assisted AddressSanitizer". 2017. https://clang.llvm.org/docs/ HardwareAssistedAddressSanitizerDesign.html. [Online; accessed 03-31-2019].

[47] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. 2012. ILR: Where'd my gadgets go?. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 571–585.

[48] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 1–11.

[49] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*. 177–192.

[50] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1470–1486.

[51] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 969–986.

[52] Intel. 2019. Control-flow Enforcement Technology Preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. Last accessed March 24, 2020.

[53] Introduction to Intel® Memory Protection Extensions. 2013. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html. [Accessed 03-24-2020].

[54] Kyriakos K Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1868–1882.

[55] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. 2016. "The Web/Local" Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 791–804.

[56] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 339–348.

[57] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 461–477.

[58] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*. 205–221.

[59] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 147–163.

[60] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2018. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. 81–116.

[61] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. 2013. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 721–732.

[62] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 276–291.

[63] LBNL-FTP-PKT. 2004. Anonymous FTP connections dataset at the Lawrence Berkeley National Laboratory. https://ee.lbl.gov/anonymized-traces.html. [Online; Accessed 03-25-2020].

[64] Intel(R) Processor Trace Decoder Library. 2013. https://github.com/intel/libipt. Accessed April 03, 2020.

[65] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*. 177–194.

[66] Chen Liu, Zhiliu Yang, Zander Blasingame, Gildo Torres, and James Bruska. 2018. Detecting Data Exploits Using Low-level Hardware Information: A Short Time Series Approach. In *Proceedings of the First Workshop on Radical and Experiential Security*. ACM, 41–47.

[67] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 529–540.

[68] LLVM. 2003. The LLVM Compiler Infrastructure. http://llvm.org/. [Accessed 04-25-2020].

[69] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 280–291.

[70] Yuan Luo, Ya Xiao, Long Cheng, Guojun Peng, and Danfeng Daphne Yao. 2020. Deep Learning-Based Anomaly Detection in Cyber-Physical Systems: Progress and Opportunities. arXiv:2003.13213

[71] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 115–128.

[72] Sparsh Mittal. 2018. A Survey of Techniques for Dynamic Branch Prediction. *CoRR* abs/1804.00261 (2018). arXiv:1804.00261 http://arxiv.org/abs/1804.00261

[73] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2018. Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 167–182.

[74] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–258.

[75] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 128–139.

[76] National Vulnerability Database (NVD). 2006. ProFTPD remote exploit. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5815. [Online; accessed 03-25-2020].

[77] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A. Sadeghi. 2019. HardScope: Hardening Embedded Systems Against Data-Oriented Attacks. In *56th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[78] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. *SIGMETRICS Perform. Eval. Rev.* 46, 1 (June 2018), 111–112.

[79] Aleph One. 1996. Smashing the Stack for Fun and Profit. *Phrack* 7, 49 (November 1996). http://www.phrack.com/issues.html?issue=49&id=14

[80] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 144–164.

[81] Dan Pelleg and Andrew W. Moore. 2000. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *ICML*.

[82] Alexander Peslyak. 1997. “return-to-libc” attack. *Bugtraq, Aug* (1997).

[83] J. Pewny, P. Koppe, and T. Holz. 2019. STEROIDS for DOPed Applications: A Compiler for Automated Data-Oriented Programming. In *IEEE European Symposium on Security and Privacy (Euro S&P)*. 111–126.

[84] Qualcomm Technologies Inc. 2017. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[85] Prabhu Rajasekaran, Stephen Crane, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. CoDaRR: Continuous Data Space Randomization against Data-Only Attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 494–505.

[86] S. Rapps and E. J. Weyuker. 1985. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering* SE-11, 4 (1985), 367–375.

[87] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)* 15, 1 (2012), 1–34.

[88] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. 2017. Revisiting browser security in the modern era: New data-only attacks and defenses. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 366–381.

[89] SAFE secure computing platform. 2019. http://www.crash-safe.org/. [Accessed 08-12-2019].

[90] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. 2014. Modular protections against non-control data attacks. *Journal of Computer Security* 22, 5 (2014), 699–742.
[91] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium*. 25–41.
[92] Edward J Schwartz, Cory F Cohen, Jeffrey S Gennari, and Stephanie M Schwartz. 2020. A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1789–1801.
[93] Jeff Seibert, Hamed Okhravi, and Eric Söderström. 2014. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 54–65.
[94] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIXATC 12)*. 309–318.
[95] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security*. 552–561.
[96] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. 2015. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 401–413.
[97] Xiaokui Shu, Danfeng Yao, Naren Ramakrishnan, and Trent Jaeger. 2017. Long-span program behavior modeling and attack detection. *ACM Transactions on Privacy and Security (TOPS)* 20, 4 (2017), 1–28.
[98] Xiaokui Shu, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Trent Jaeger. 2017. Long-Span Program Behavior Modeling and Attack Detection. *ACM Transactions on Privacy and Security* 20, 4 (Sept. 2017), 1–28.
[99] Kanad Sinha and Simha Sethumadhavan. 2018. Practical memory safety with REST. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 600–611.
[100] SIR. 2006. Software-artifact Infrastructure Repository. http://sir.unl.edu/. [Accessed 04-25-2020].
[101] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 574–588.
[102] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17.
[103] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1275–1295.
[104] Z. Sun, B. Feng, L. Lu, and S. Jha. 2020. OAT: Attesting Operation Integrity of Embedded Devices. In *2020 IEEE Symposium on Security and Privacy (SP)*.
[105] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 48–62.
[106] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 256–267.
[107] PaX Team. 2003. PaX address space layout randomization (ASLR). (2003).
[108] Testing Exploitable Buffer Overflows From Open Source Code. 2018. https://samate.nist.gov/SRD/view.php?tsID=88. [Online; accessed 01-08-2018].
[109] The Rust Programming Language. 2019. https://www.rust-lang.org/. [Accessed 08-12-2019].
[110] Gildo Torres and Chen Liu. 2016. Can data-only exploits be detected at runtime using hardware events?: A case study of the heartbleed vulnerability. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2.
[111] Stylianos Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2017. Towards automatic compartmentalization of C programs on capability machines. In *Workshop on Foundations of Computer Security 2017*. 1–14.
[112] Julien Vanegue. 2013. The Automated Exploitation Grand Challenge. https://openwall.info/wiki/_media/people/jvanegue/files/aegc_vanegue.pdf. Last accessed Feb 12, 2020.
[113] Kuznetsov Volodymyr, Szekeres Laszlo, Payer Mathias, Candea George, and R Sekar. 2014. Code-pointer integrity. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[114] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM Symposium on Operating Systems Principles*. 203–216.
[115] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications*

security. ACM, 157–168.

[116] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. 2015. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 20–37.

[117] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 367–382.

[118] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 457–468.

[119] Jidong Xiao, Hai Huang, and Haining Wang. 2015. Kernel data attack is a realistic security threat. In *International Conference on Security and Privacy in Communication Systems*. Springer, 135–154.

[120] Danfeng Yao, Xiaokui Shu, Long Cheng, and Salvatore J. Stolfo. 2017. Anomaly detection as a service: challenges, advances, and opportunities. *Synthesis Lectures on Information Security, Privacy, and Trust* 9, 3 (2017), 1–173.

[121] Danfeng (Daphne) Yao, Xiaokui Shu, Long Cheng, and Salvatore J. Stolfo. 2017. Anomaly Detection as a Service: Challenges, Advances, and Opportunities. *Synthesis Lectures on Information Security, Privacy, and Trust* 9, 3 (2017), 1–173.

[122] Suan Hsi Yong and Susan Horwitz. 2003. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 307–316.

[123] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 559–573.

[124] Hao Zhang, Danfeng Daphne Yao, Naren Ramakrishnan, and Zhibin Zhang. 2016. Causality reasoning about network events for detecting stealthy malware activities. *Computers & Security* 58 (2016), 180–198.

[125] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*. 337–352.

[126] X. Zhuang, T. Zhang, and S. Pande. 2006. Using Branch Correlation to Identify Infeasible Paths for Anomaly Detection. In *IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 113–122.