Exploitation Techniques and Defenses for Data-Oriented Attacks

Long Cheng*, Hans Liljestrand†, Md Salman Ahmed‡, Thomas Nyman†, Trent Jaeger§, N. Asokan†, and Danfeng (Daphne) Yao‡
*School of Computing, Clemson University, USA
†Department of Computer Science, Aalto University, Finland
‡Department of Computer Science, Virginia Tech, USA
§Department of Computer Science and Engineering, Pennsylvania State University, USA

***Abstract*—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 systematization of knowledge (SoK) paper, we first map data-oriented exploits, including Data-Oriented Programming (DOP) and Block-Oriented Programming attacks, to their assumptions/requirements and attack capabilities. We also compare known defenses against these attacks, in terms of approach, detection capabilities, overhead, and compatibility. Then we discuss the possible frequency anomalies of data-oriented attacks, especially the frequency anomalies of DOP attacks with experimental proofs. It is generally believed that control flows may not be useful for data-oriented security. However, the frequency anomalies show that data-oriented attacks (especially DOP attacks) may generate side-effects on control-flow behavior in multiple dimensions. In the end, we discuss challenges for building deployable data-oriented defenses and open research questions.**

***Index Terms*—Data-oriented attacks; Exploitation techniques; Defenses; Systematization of knowledge (SoK);**

I. Introduction

Memory-corruption vulnerabilities are one of the most common attack vectors used to compromise computer systems. Such vulnerabilities can be exploited in different ways, which potentially allow attackers to perform arbitrary code execution and data manipulation. Existing memory corruption attacks can be broadly classified into two categories: *i)* control-flow attacks [1], [2], [3] and *ii)* data-oriented attacks (also known as non-control data attacks) [4], [5], [6], [7], [8]. Both types of attacks can cause significant damages to a victim system [9].

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 [1], code reuse [2], and Return-Oriented Programming (ROP) [3]. To counter these attacks, many defense mechanisms have been proposed, such as stack canaries [10], Data Execution Prevention (DEP) [11], Address Space Layout Randomization (ASLR) [12], Control-Flow Integrity (CFI) [13], Return-Flow Guard (RFG) [14], Intel's CET [15] and MPX [16]. In particular, CFI-based solutions [17] 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 security policies on indirect control transfer instructions (*e.g.*, ret/jmp).

In contrast to control-flow attacks, data-oriented attacks [5], [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 or private keys); 2) privilege escalation (*e.g.*, by manipulating user identity data) [5]; 3) performance degradation (*e.g.*, resource wastage attack) [19]; and 4) bypassing security mitigation mechanisms [20].

As launching control-flow attacks becomes increasingly difficult due to many deployed defenses against control-flow hijacking, data-oriented attacks[1] are likely to become an appealing attack technique for system compromise [6], [7], [8], [20], [21], [22]. 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 [23]. For example, arbitrary code-execution attacks are possible if an attacker could corrupt parameters of system calls (*e.g.*, `execve()`) [9]. Recently, Hu *et al.* [7] proposed Data-Oriented Programming (DOP), a systematic technique to construct expressive (*i.e.*, Turing-complete) non-control data exploits. Ispoglou *et al.* [18] presented the Block-Oriented Programming (BOP), a code reuse technique 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 systematization of knowledge (SoK) papers [4], [24],

[1]In this work, we mainly focus our investigation on data-oriented attacks that are caused by memory-corruption vulnerabilities. Data-only attacks that are caused by hardware transient faults or logic errors in code are beyond the scope of this work.

[25] related to memory corruption vulnerabilities, our work specifically focuses on data-oriented attacks. In addition to generic memory corruption prevention mechanisms discussed in [4], [24], [25] 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 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 [6] 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 Compiler (BOPC) demonstrated in [18], fundamental differences between DOP and BOP, and flexibility issues in exploit writing languages such as MINDOP and SPloit Language (SPL). We also discuss representative data-oriented exploits including their assumptions/requirements and attack capabilities (Section II).
* We present a three-stage model for data-oriented attacks and discuss defense techniques according to different stages. Then, we provide a comparative analysis of recent defensive approaches focusing on data-oriented attacks (Section III).
* We investigate possible side-effects of data-oriented attacks, particularly the side-effects of DOP attacks on control-flow behaviors in multiple dimensions. We show the proof of frequency anomalies caused by DOP attacks (Section IV).

We also discuss some open research problems and unsolved challenges (Section V).

II. Data-oriented Attacks

In this section, we first describe why data-oriented attacks has received attention among security researchers in recent years. Then we introduce two categories of exploitation techniques to launch data-oriented attacks (Section II-B). We reproduce a real-world DOP attack against the ProFTPD FTP-server [7] and present a detailed description of the attack to demonstrate how the complex attack achieves rich expressiveness (Section II-C). Then, we map representative data-oriented exploits in the literature to their assumptions/requirements and attack capabilities (Section II-E).

A. *Why do data-oriented attacks receive attention?*

The war games of memory corruption between attackers and defenders started in the mid-'90s and have been continuing to date. The stack smashing attack [71] and related attacks such as overwriting Structured Exception Handler (SEH) [72] were popular among attackers for a few years. In those attacks, attackers inject malicious code to the stack or heap and then redirect the control-flow to the injected code. Data Execution Prevention (DEP) [11] and Structured Exception Handling Overwrite Protection (SEHOP) [73] render those attacks impossible.

A new class of attacks, namely the code-reuse attacks, dominated in the last decade due to their capability of bypassing DEP. Code-reuse includes attacks such as return-to-libc [74], ROP [75], Call-Oriented Programming [76], and Jump-Oriented Programming [77]. ASLR [78] was introduced to make code-reuse attacks difficult and unreliable. However, attackers demonstrated several methods for bypassing ASLR by exploiting information leakage vulnerabilities [79], [80], [81], [82]. An information leak vulnerability may render the ASLR defense ineffective because it allows inferring the layout of the address space of a process. A few influential code-reuse attacks include AOCR [83], JIT-ROP [84], BROP [85], CFB [86], OOC [76], and EHH [87]. Most of these attacks extract ROP gadgets from a process' code segment by utilizing an information leak and then chain the gadgets together to bypass ASLR- and CFI-related defenses. A gadget is a short sequence of instructions (usually 2-5 instructions per gadget) that performs a single operations such as loading a constant to a register or reading a value from memory. The prevalence of information leakage vulnerabilities motivated researchers to develop leakage-resilient defenses. The existing leakage-resilient defenses are broadly in four categories: *i)* fine-grained code randomization, *ii)* re-randomization, *iii)* memory protection, and *iv)* code/data pointer integrity.

Researchers have put significant efforts on developing practical control-flow integrity [63] defenses such as CCFIR [66] and bin-CFI [65]. However, software supported CFI and shadow stack implementations incur significant slowdown, primarily due to the enforcement of control-flow transfers at runtime after constructing a non-trivial control-flow graph [88], [89], [90], [91]. This non-trivial control-flow graph approximates the set of control-flow transfers. Researchers often balance the trade-offs between security and practicality of CFI-based solutions by relaxing the CFI policies [65], [66], [92]. But CFI solutions that relax CFI policies may not prevent some code-reuse attacks [87]. Thus, in recent years, researchers' focus has been transferred to the hardware supported CFI [67], [68], [69], [70], [93], [94], [95], [96] and shadow stack [68], [97], [98] implementations that improve performance without relaxing CFI policies. Budiu *et al.* first demonstrated hardware-assisted CFI in [93]. The overall goal of hardware-assisted CFI defenses is to embed unique labels in indirect branch instructions and enforce control-flow integrity at each execution of an indirect branch instruction by loading those labels in dedicated registers using CFI instructions. CFI defenses (software- and hardware-based) make most code-reuse attacks unreliable. Due to this fact, many attackers have shifted their focus on data-oriented attacks in recent years [7], [18]. However, that does not mean that code-reuse attacks have completely prevented in the presence of CFI. For example, Veen *et al.* [99] demonstrated code-reuse attacks in the presence of CFI, leakage-resistant code randomization, and code-pointer integrity.

In addition to CFI defenses, Table I summarizes other generic defenses against memory corruption attacks, including memory safety, software compartmentalization, leakage-

Defense Name and Year Published	Defense Type	Protection From
PointGuard (2003) [26], Data integrity (2003, 2007) [27], [28], HardBound (2008) [29], SoftBound (2009) [30], AddressSanitizer (2012) [31], Code-Pointer Integrity (2014) [32], Intel MPX (2015) [33], Low-fat-pointers (2016-17) [34], [35], Fat-pointer (2017) [36], Pointer Authentication (2017) [37], Hardware-assisted AddressSanitizer (2018) [38]	Memory safety	Security vulnerabilities dealing with memory accesses
SFI (1994) [39], XFI (2006) [40], BGI (2009) [41], LXFI (2011) [42], CHERI (2014) [43]	Software compartmentalization	Consequences of attacks that leverage memory vulnerabilities, including software compartmentalization and randomization (diversification) techniques
ASLP (2006) [44], ILR (2012) [45], Binary stirring (2012) [46], Multicompiler (2013-14) [47], [48], Selfrando (2016) [49], CCR (2018) [50]	Fine-grained Randomization	Basic ROP exploits
TASR (2015) [51], Shuffler (2016) [52], Remix (2016) [53], Runtime-ASLR (2016) [54]	Re-randomization	JIT-ROP and BROP type ROP attacks if re-randomization time is less than attack time
XnR (2012) [55], HideM (2015) [56], Readactor (2015) [57], Heisenbyte (2015) [58], NEAR (2016) [59]	Memory protection	JIT-ROP and BROP type ROP attacks
PointGuard (2010) [60], Oxymoron (2014) [61], ASLR-Guard (2015) [62]	Code pointer integrity	Code pointer overwrite or corruption
CFI (2005) [63], ROP-Guard (2012) [64], Bin-CFI (2013) [65], CCFIR (2013) [66]	Software-assisted CFI	Code reuse and code injection attacks
CFIMon (2012) [67], Branch regulation (2012) [68], Hcfi (2016) [69], HAFIX (2015) [70]	Hardware-assisted CFI	Code reuse and code injection attacks

TABLE I. A set of recent and influential memory safety, software compartmentalization, leakage-resilient and CFI defenses

resilient defenses, which can also be applied to mitigate data-oriented attacks. However, memory corruption problems are still possible due to the lack of deployable solutions in terms of both effectiveness and efficiency [4]. In Sec. III-B, we provide a detailed discussion of representative generic defenses.

B. *Classification of data-oriented attacks*

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

1) DDM refers to a category of attacks in which an attacker directly manipulates the target data to accomplish the malicious goal. It requires the attacker to know the precise memory address of the target non-control-data. The address 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) or by reusing the runtime randomized address stored in memory [6]. Several types of memory corruption vulnerabilities, *e.g.*, format string vulnerabilities, buffer overflows, integer overflows, and double free vulnerabilities [25], allow attackers to directly overwrite memory locations within the address space of a vulnerable application. Chen *et al.* [5] 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 example of the attack on decision-making data in SSH server, which was first reported in [5]. A local flag variable `authenticated` is used to indicate 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 is able to corrupt the `authenticated` variable to a non-zero value, which bypasses the user authentication (line 16).

```
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 [5]

2) DOP is an advanced technique to construct expressive non-control data exploits [7]. It allows an attacker to perform arbitrary computations in program memory by chaining the execution of short sequences of instructions (referred to as *data-oriented or DOP gadgets*). DOP gadgets are similar to ROP gadgets that can perform arithmetic/logical, assignment, load, store, jump, and conditional jump operations. The idea of DOP is to reuse DOP gadgets for malicious purposes other than the developer's original intent. Similarly, Block-Oriented Programming (BOP) [18] constructs exploit programs by chaining BOP gadgets without violating CFI, where each BOP gadget corresponds to a basic block that contains a DOP gadget. 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 chains DOP 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 II-C.

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.* [8] recently demonstrated a multi-step DDM with Nginx (listed in Table II). 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) [8].

Like the DOP attack, a multi-step DDM attack violates data-flow integrity. DDM is a pre-requisite for DOP. However, DOP is much more complex than the multi-step DDM. We summarize their key differences in the following.

* **Gadgets and code reuse.** DOP/BOP attacks involve reusing code execution through CFI-compatible gadgets. Multi-step DDM hinges on direct memory writes and does not involve any gadget executions.
* **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).

A significant contribution by Ispoglou *et al.* in [18] is the block-oriented programming compiler (BOPC). BOPC is the first compiler technique that automates the BOP/DOP attack generation (given the arbitrary memory write vulnerability). With the automatically generated attack payloads by the compiler, an attacker first performs a series of DDMs to modify memory and then launches a BOP/DOP attack by chaining gadgets that leverage memory manipulation via DDMs.

C. *Demystifying the ProFTPd DOP attack*

We use the ProFTPd DOP attack crafted by Hu *et al.* [7] to illustrate the typical flow of DOP attacks. The goal of this DOP attack is to bypass randomization defenses (such as ASLR [12]), 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 de-referencing 7 times within the server's memory space.

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 [100]). The overflow can be exploited by an attacker to obtain an arbitrary write primitive. The server program provides a feature to display customized messages when a user enters a directory. The 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 2: The vulnerable function in ProFTPd

Listing 2 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

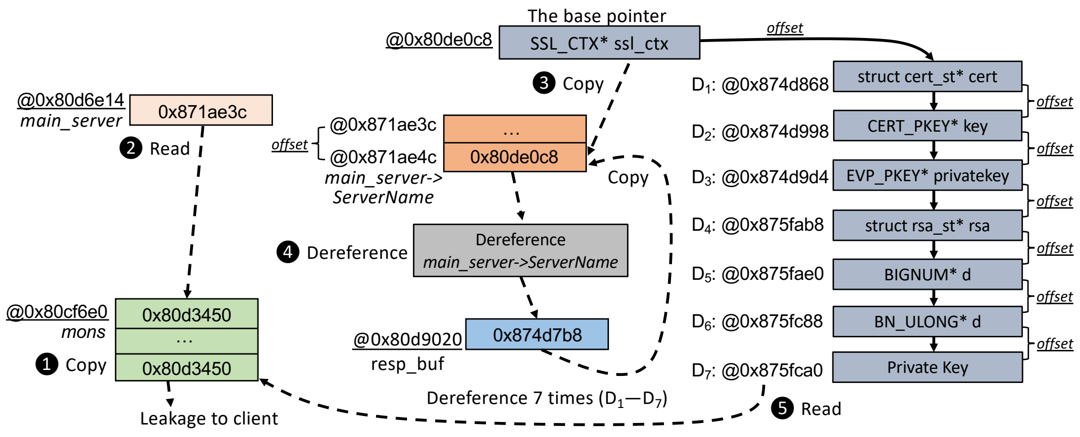

Fig. 1: ProFTPd DOP attack flow. An attacker needs to know the underlined addresses and offsets to launch the attack.

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) 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 2). We reproduced the ProFTPd DOP attack, and observed that the vulnerable function sreplace is called over 180 times during the attack.

Fig. 1 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 Fig. 1, 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 Fig. 1, 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 currently located at main_server->ServerName, by triggering the execution of the dereference gadget in line 4 of Listing 2. 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 Fig. 1) 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 ($D_1 \sim D_7$ in Fig. 1) following the dereference chain as shown in Fig. 1. 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.

D. Block-Oriented Programming (BOP) attack

The core of the Block-Oriented Programming (BOP) [18] attack is the Block-Oriented Programming Compiler (BOPC) that automates the process of constructing data-oriented exploits. BOPC provides the flexibility to construct data-oriented exploits by defining the exploit goals using a high-level C like language called SPloit Language (SPL). BOPC is the compiler of an SPL exploit. The target binary and its program states are the execution environment. Fig. 2(a) shows three statements of a sample SPL payload. Each statement of the

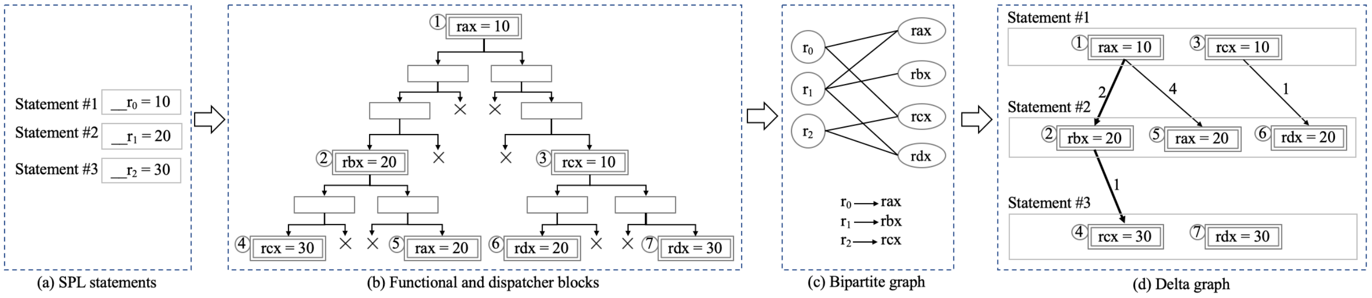

Fig. 2: Four major components of BOP Compiler. The double and single border boxes (□) indicate functional and dispatcher blocks. The number inside a circle (◯) represents the functional block number. The × represents irrelevant basic blocks.

SPL exploit or payload is mapped to a functional block and a set of dispatcher blocks (where each block is actually a basic block). The double- and single-bordered rectangles in Fig. 2(b) represent functional and dispatcher blocks. A functional block executes the semantics of an SPL statement whereas a set of dispatcher blocks links between two functional blocks.

BOPC matches a functional block to an SPL statement using constraint summaries by isolating and symbolically executing the block. Constraint summaries include registers, memory locations, jump types, and library calls. BOPC searches for functional blocks for SPL statements and selects a functional block for each statement. To do this, BOPC creates a bipartite graph by associating each SPL statement with a set of functional blocks that may potentially serve the SPL statement. Fig. 2(c) shows a bipartite graph for the SPL statements in Fig. 2(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. Fig. 2(c) shows one such association ($r_0 \rightarrow rax$, $r_1 \rightarrow rbx$, and $r_2 \rightarrow rcx$).

However, this selection process is not arbitrary because the functional blocks are volatile. Due to the volatility characteristics of functional blocks, the selection of a functional block may reduce the availability of other functional blocks. This volatility behavior makes the functional block selection process an NP-hard problem. Thus, the selection process requires heuristics for selecting a set of suitable functional blocks in polynomial time.

As a set of dispatcher blocks links between two functional blocks, an arbitrary selection of the dispatcher blocks may also clobber the SPL state. This is why a small set of dispatcher blocks is more likely to be non-clobbering than a large set of dispatcher blocks for linking two functional blocks. To select a small set of non-clobbering dispatcher blocks, BOPC constructs a delta graph. 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. Fig. 2(d) shows a delta graph for the SPL statements in Fig. 2(a). A recursive version of Dijkstra's [101] 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. Fig. 2(d) indicates the sub-graph using bold arrows.

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 [102]. 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.

Difference between DOP and BOP. Though Data-Oriented Programming (DOP) is a generalization of data-oriented attacks, Block-Oriented Programming (BOP) has a few key differences from DOP: *i)* automatic generation of exploit payload, *ii)* exploits are written in the easily understandable high-level C-like SPL, and *iii)* granularity. In DOP, one must analyze and construct exploit manually. However, one can write exploit using SPL by specifying the exploit goal. BOPC automatically parses the SPL exploit and constructs BOP gaget chain automatically. In terms of the granularity, DOP works at the gadget level whereas BOP works at the basic block level. Another noticeable thing is that BOPC is not fully automated. The output produced by BOPC is a set of address-value pairs. An attacker requires to modify an address with the corresponding value from the address-value pairs in order to launch an attack. Thus, BOPC is not designed to implement a complete end-to-end attack.

Exploit payloads can be specified using MINDOP for DOP attacks whereas SPL is used for BOP exploits. Both MINDOP and SPL are Turing-Complete languages. That means both languages support memory read/write, assignment, arithmetic operation, logical operation, control-flow (*e.g.*, jump), function call, and system API calls. The fundamental differences between the two languages are granularity and flexibility as discussed below.

SPL: SPL is an exploit writing language based on C dialect to express data-oriented attacks. This language provides the flexibility of expressing BOP gadgets in a high-level language like C which is easy to understand from an exploit point of view. SPL also allows the explicit access of virtual registers, library functions, and APIs to call OS functions. The BOPC compiler translates each statement of an SPL program to a set of constraints. The set of constraints are then solved to

Targeted Application and Year	Type	Assumption/Requirement	Capability
Chrome [103], 2016	DDM	Identified security-critical variables, and arbitrary read/write capability	Bypass the same-origin policy
Linux Page Table [104], 2017	DDM	Kernel code writable, and arbitrary read/write capability	Bypass the kernel CFI
InternetExplorer, Chrome [105], 2017	DDM	Identified security-relevant variables, and arbitrary read/write capability	Information leakage, bypass the same origin policy, etc.
Nginx [8], 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.
ProFTPd [7], 2016	DOP	Memory addresses of multiple involved data, identified gadgets/dispatchers, and arbitrary read/write capability	Private key leakage w/ ASLR
ProFTPd, Nginx, sudo, etc. [18], 2018	BOP	Arbitrary read/write primitive	Automatic construction of BOP gadget chain or exploit payload

TABLE II. Recent data-oriented attacks pose serious threats against real-world programs.

map the statement to a set of basic blocks that satisfy the constraints. Though SPL is Turing complete, it is still quite limited in terms of its richness of operations.

MINDOP: MINDOP is a simple language with a set of virtual instructions and virtual register operands. MINDOP supports 6 kinds of virtual instructions: *i)* arithmetic/logical, *ii)* assignment, *iii)* load, *iv)* store, *v)* jump, and *vi)* conditional jump. Each instruction of MINDOP can be simulated using x86 instructions.

E. Representative data-oriented attacks

In the seminal work of non-control data attacks [5] and later FlowStitch [6], the authors have described more than 20 different data-oriented exploits (most of them are single-step DDM attacks). More recently, several research efforts have shown that data-oriented attacks pose serious threats to real-world programs.

Jia *et al.* [103] 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.* [104] 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.* [105] introduced a new technique, called memory cartography, that an adversary can use to navigate itself 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.* [8] 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, it is possible that an attacker re-configures 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 at runtime 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 [8] simulated the arbitrary write vulnerability in the recent version of Nginx, rather than exploiting a real-world vulnerability.

Table II summarizes these recent data-oriented attacks. Because existing CFI-based solutions are rendered defenseless under data-oriented exploits, such threats are particularly alarming. To construct a data-oriented exploit, attackers must have an 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.

III. Defenses Against Data-Oriented Attacks

In this section, we first describe a three-stage model for data-oriented attacks, and a taxonomy of existing applicable defense techniques. Then, we provide a comparative analysis of recent defensive approaches, particularly against data-oriented attacks.

A. Three-stage model for launching data-oriented attacks

Fig. 3 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$\rightarrow$S1 and S2$\rightarrow$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 for DDM exploits.

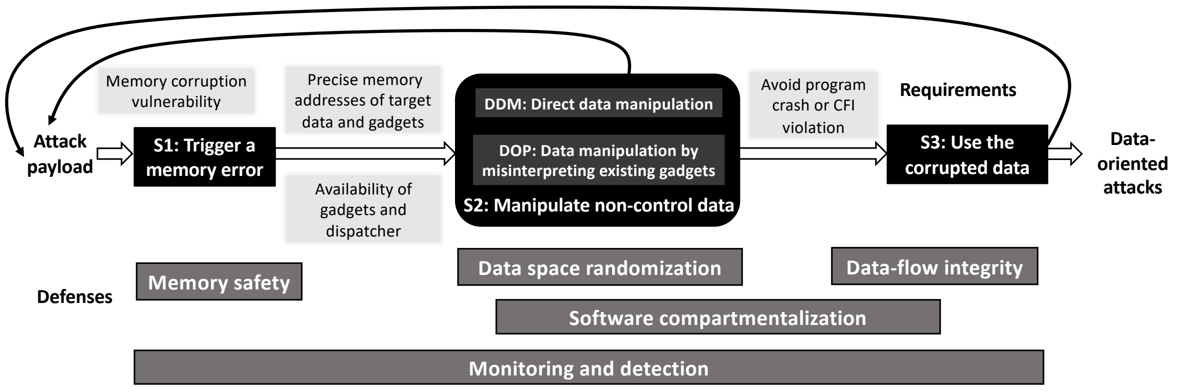
Memory corruption vulnerability
Precise memory addresses of target data and gadgets
Availability of gadgets and dispatcher
Avoid program crash or CFI violation
Requirements
Attack payload
S1: Trigger a memory error
DDM: Direct data manipulation
DOP: Data manipulation by misinterpreting existing gadgets
S2: Manipulate non-control data
S3: Use the corrupted data
Data-oriented attacks
Defenses
Memory safety
Data space randomization
Data-flow integrity
Software compartmentalization
Monitoring and detection

Fig. 3: Stages in data-oriented attacks and mitigation in different stages

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

* Knowing the exact *location of target non-control data in memory*. Due to the wide deployment of exploit mitigation technologies such as DEP and ASLR, it is likely attackers need to first leverage memory disclosure vulnerabilities to circumvent the address space randomization [8]. 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 *transformation of an attack payload to the impact on memory space* of the target program. For example, a continuous buffer overflow 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 by the memory corruption vulnerability, and triggerable by the attack payload.

* *Stitchability of disjoint DOP gadgets*. A gadget dispatcher is needed to dispatch and execute 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.

B. Taxonomy of applicable defense techniques

We briefly discuss 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 [4], [24], [25].

1) S1 Defense – Preventing exploitation of memory errors: Memory safety enforcement is the first line of defense, which aims to prevent both spatial and temporal memory errors, such as buffer overflows and use-after-free errors. Memory-safe programming languages achieve this with built-in runtime bounds checks and garbage collection that make them immune to memory errors. In contrast, *memory-unsafe* languages such as C/C++ lack built-in memory safety guarantees. Programs written in memory-unsafe languages therefore commonly exhibit memory errors that may make them vulnerable to runtime exploitation. Enforcing all memory accesses staying within the bounds of intended objects would completely eliminate the pre-conditions for attacks that rely on gaining access to a prohibited area of memory. Despite considerable prior research in retrofitting memory-unsafe programs with memory safety guarantees, memory-safety problems persist due to an trade-off between effectiveness and efficiency: low-overhead approaches usually offer inadequate protection/coverage, while comprehensive solutions either incur a high performance overhead or provide limited backward compatibility [4], [106].

SoftBound [107] and *HardBound* [108] perform pointer bounds checks against metadata stored in a shadow memory area. 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 around 10%. Both schemes have a worst-case memory overhead of 200%. *Fat-pointer* schemes store the associated bounds metadata [109] together with pointers, *e.g.*, by increasing their length [110] or by borrowing unused bits from pointers [109]. But changing the representation of pointers in memory breaks both binary and source code compatibility. *Code-Pointer Integrity* (CPI) [111] provides control-flow hijacking protection with a very low performance overhead (*e.g.*, 8.4% slowdown for C/C++ program). However, it only focuses on code-pointer checking without providing the complete memory safety.

2) S2 Defense – Providing a barrier to access to data or guess memory layout: The purpose of S2 defenses is to mitigate the consequences of attacks in the presence of memory vulnerabilities. S2 defenses include *software compartmentalization* [112], [113], [114] and *address space or data layout randomization* [12], [115] techniques. They serve as the second line of defense, which creates a barrier for attackers trying to access target data or guess the memory layout.

Software compartmentalization isolates software components into distinct protection domains in order to limit the utility of existing memory errors (*i.e.*, when the memory error and data to be manipulated exist in different protection domains), but also limit the abilities of a compromised software component. For example, *Software Fault Isolation* (SFI) [112] compartmentalizes software in a single address space by sandboxing untrusted modules into separate fault domains. This compartmentalization ensures that code in the fault domain is unable to directly access memory or jump to code outside the reserved portion of address space, but must interact with code outside it's domain through well-defined call interfaces.

Randomization aims to hide attack targets by randomizing the location of program segments [24], layout of the code [116], layout of data [115] or the data itself [117] so that unauthorized access would lead to unpredictable behavior. In particular, data space randomization [115], [118], [119] aims to randomize the representation of data stored in program memory at runtime to make it unpredictable for unauthorized accesses, and thus reducing the possibility that attackers can leak security-critical memory addresses or manipulate the content of targeted data. ASLR [12] randomly chooses the base addresses of the stack, heap, code segment, and shared libraries. Data Space Randomization (DSR) [115] encrypts data stored in memory, rather than randomizing the location. 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 [4]. On the other hand, information leaks can undermine randomization techniques. In addition, data/address encryption based solutions are not binary compatible (*i.e.*, protected binaries are incompatible with unmodified libraries) [4].

3) S3 Defense – Preventing/detecting use of corrupted data: Data-Flow Integrity (DFI) [122] 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 (*i.e.*, for each value read instruction, it statically computes the set of write instructions that may write the value). However, DFI usually overestimates the set of valid write instructions since the set is statically determined without runtime information. Moreover, Software-based DFI incurs a high performance overhead [7] 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 [122]. Hardware-based DFI, *e.g.*, HDFI [121], is efficient, but limited by the number of simultaneous protection domains it can support.

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.

Szekeres *et al.* [4] provide a systematic overview of memory corruption attacks and mitigations. They highlighted that though a vast number of solutions have been proposed, memory corruption attacks continue to pose a serious security threat. Real-world software exploits are still possible because currently deployed defenses can be bypassed. Program anomaly detection complements the aforementioned mitigation techniques, and serves as the last line of defense against data-oriented attacks. As shown in Fig. 3, passive monitoring based program anomaly detection has the potential to detect anomalous program behaviors exhibited in all the three stages of data-oriented attacks.

C. Defense mechanisms against data-oriented attacks

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

YARRA [21] is a C language extension that validates a pointer's type for *critical data types* annotated by developers, which is an S1 defense. 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 a performance overhead in the order of 400%∼600%. In addition, YARRA relies on the programmers' manual annotations, which is undesirable for complicated programs.

HardScope [22] is a hardware-assisted variable scope enforcement approach to mitigate data-oriented attacks by introducing 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.* [22] demonstrated the effectiveness of HardScope for the RISC-V open instruction set architecture, by introducing a set of seven new instructions. HardScope instructions are instrumented at compile-time, and memory access constraints are enforced at runtime. It shows that HardScope has a real-world performance overhead of 3.2% in embedded benchmarks. Although HardScope significantly reduces the usefulness of DOP gadgets and thwarts Hu *et al.* [7]'s example attacks, HardScope cannot guarantee the absence of DOP gadgets in arbitrary programs.

PrivWatcher [120] 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. It involves a set of kernel modifications including relocating process credentials into a safe region, code instrumentation and runtime data integrity verification, in order to provide non-bypassable

Defense and Year	Stage	Approach	Security Guarantee	Overhead	General Approach
YARRA [21], 2011	S1 (Pointer safety)	Program instrumentation	User-specified critical data	400%∼600% (whole program)	✓
HardScope [22], 2018	S2 & S3 (Compartmentalization)	Hardware extension	Context-specific memory isolation	∼3.2%	✗
PrivWatcher [120], 2017	S2 (Compartmentalization)	Kernel modification	Protect process credentials data in Linux kernel	∼3% (94% in extreme cases)	✗
HDFI [121], 2016	S2 (Compartmentalization)	Hardware extension	Coarse-grained data-flow isolation	∼2%	✗
PT-Rand [104], 2017	S2 (Randomization)	Kernel modification	Protect kernel page tables	0.22%	✗
DFI [122], 2006	S3	Program instrumentation	Data-flow integrity	∼100%	✓
CVI [123], 2018	S3	Program instrumentation	Selective data-flow integrity	∼2.7%	✗
Hardware-based detector I [124], 2016	S1 – S3	Hardware performance counters	Detection of Heartbleed attacks with an accuracy of 70% to 92% and 1% false negatives	NR	✗
Hardware-based detector II [125], 2018	S1 – S3	Hardware performance counters	Detection of Heartbleed attacks with an accuracy of 97.75% to 98.36%	NR	✗

TABLE III. Comparison of defensive mechanisms against data-oriented attacks. NR means not reported.

integrity assurances. It ensures the Time of Check To Time of Use (TOCTTOU) consistency between verification and usage contexts for process credentials by adopting a dual reference monitor model. The authors implemented the PrivWatcher prototype in Ubuntu Linux. The experiment results show that PrivWatcher incurs an overhead less than 3%. But it incurs more than 94% overhead for applications that involve installing new `task_struct` structures to processes.

Hardware-Assisted Data-flow Isolation (HDFI) [121] 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.

Davi *et al.* [104] presented a data-oriented attack against kernel page tables to bypass CFI-based kernel hardening techniques, and subsequently attackers can execute arbitrary code with kernel privileges. To mitigate the threat, they proposed *PT-Rand*, which randomizes the location of page tables to prevent attackers from manipulating page tables by means of data-oriented attacks. Evaluation results show that PT-Rand on Debian only incurs a low overhead of 0.22% for common benchmarks. However, it is still possible attackers undermine these schemes if the secret information (*e.g.*, randomization secret) is leaked or inferred [22].

CVI (Critical Variable Integrity) [123] verifies define-use consistency of critical variables for embedded devices. The define-use consistency is defined as the property 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 [122], CVI is based on compile-time instrumentation and frequent runtime checking, which incurs a high overhead for the complete protection.

Hardware-based detector I. Torres *et al.* [124] presented a framework for detecting data-only exploits by collecting information from Hardware Performance Counters (HPCs). However, the authors mainly focused on the Heartbleed attack [126], which exploit a buffer overread vulnerability in the OpenSSL library to leak information from memory (*e.g.*, the private key associated with the website's certificate). The original intentions of designing HPC registers were to debug hardware designs and identify program inefficiency or bottlenecks during execution. HPC registers can also help identify some hardware events related to instructions retired, cache-misses suffered, and branches miss-predicted. The authors collected 12 such hardware events during the normal executions of OpenSSL and tagged these samples as good, where a sample consists of multiple hardware events. Similarly, they collected these 12 events during the executions of OpenSSL when OpenSSL executes under Heartbleed attacks. They tagged these samples as bad. Next, the authors used these samples to train and tested a multi-class support vector machine model to classify between normal and abnormal behaviors. The authors pointed out a few fundamental limitations such as instability and unreliability. Co-executing programs significantly affect HPC-based events. Also, the HPC-based events are dependent on instruction set architectures (ISAs). Thus, architecture support can improve the generalization and accuracy of such hardware-based detection models.

Hardware-based detector II. Liu *et al.* [125] extended the previous HPC-based data exploit detector [124]. In this work, the authors also collected 12 HPC events the same as the previous HPC-based work [124]. However, the events were collected as a 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. This approach is independent of the underlying applications. The authors evaluated their technique using different classification algorithms and found promising results, especially the Stacked Denoising Autoencoder and Echo State Network classifiers are around 98% accurate for detecting data-oriented exploits.

Table III compares representative data-oriented attack specific defensive mechanisms. PrivWatcher [120], HDFI [121], PT-Rand [104], and CVI [123] protect specific non-control data. HardScope [22] can protect against all DOP attacks that violate variable visibility rules at runtime. However, it requires developer assistance in certain settings. The main drawback of HarScope and other solutions based on new hardware extensions [121] is the high bar for deployment. They cannot be directly applied to protect user-space applications against general data-oriented attacks, in particular DOP attacks. On the hand, the two general approaches DFI [122] and YARRA [21] 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 primary limitation of the hardware-based detector I [124] is unreliability due to the unrelated hardware events produced by co-existing programs. This technique [124] is also dependent on ISAs which limits the generic deployability of the detector for detecting data-oriented attacks. The hardware-based detector II [125] overcomes these limitations by collecting the hardware events as a short time series by monitoring the execution of a vulnerable program before, during, and after the execution of the vulnerable section. The time series approach makes this technique less independent of the underlying applications.

IV. Frequency Anomalies of Data-Oriented Attacks

In addition to the defense mechanisms described above to detect or prevent data-oriented attacks, we compare the frequency distributions of a program's normal executions and when the program executes under data-oriented attacks. More specifically, we attempt to characterize how DOP impacts the frequency distributions of ProFTPd. We conducted two sets of comparisons, *i)* on macro-level interaction frequencies and *ii)* on micro-level control-flow frequencies.

1) ***Macro-level interaction frequencies:*** 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 II-C, an attacker needs to send a large number of FTP commands with malicious inputs to the ProFTPd server to corrupt the program memory repeatedly.

To detect 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. For frequency distributions of normal operations, we used the LBNL-FTP-PKT [127] 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 21482 FTP connections.

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 [128] 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.

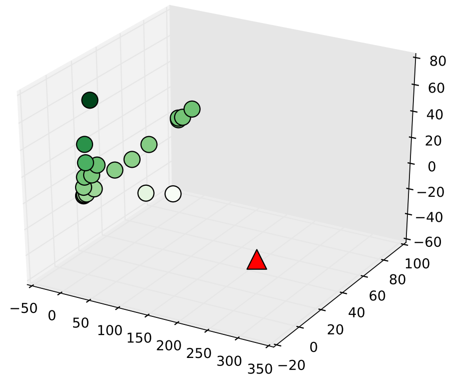

Fig. 4: For macro-level interaction frequencies, X-clustering for 2-grams of FTP commands with PCA reduction to 3-dimension using LBNL-FTP-PKT dataset [127]. The DOP attack involves an abnormally high number of client-server interactions.

Fig. 4 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.

2) ***Micro-level control-flow frequencies:*** 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.

In the ProFTPd DOP attack, an attacker crafts `.message` files (*i.e.*, as malicious payloads) to repeatedly fill up the allocated buffer and write bytes beyond the buffer in `sreplace`, which exhibits anomalous behaviors of control-flow transfers. We defined all control-flow transfers in each `sreplace` invocation as a behavior instance, following the approach in [129]. 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.

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

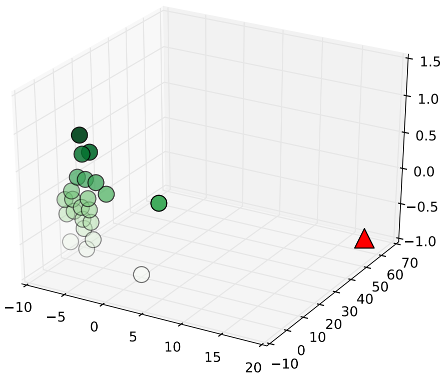

Fig. 5: For micro-level control-flow frequencies, X-clustering for 2-gram control-transfers with PCA reduction to 3-dimension in `sreplace`. The DOP attack exhibits a unique pattern of control-flow transfers in comparison to baseline executions.

in `sreplace` is shown in Fig. 5. The baseline dataset is clustered into 23 clusters. Similar to Fig. 4, the DOP instance is an obvious outlier.

BOP is likely to have a similar type of frequency anomalies because BOPC utilizes multiple arbitrary memory read/write primitives to modify the memory state of the target binary. In addition to that BOPC may trigger the arbitrary memory read/write primitives multiple times. However, the frequency anomalies may not be as observable as DOP because the frequency anomalies for BOPC largely depend on a specific attack and the nature of the vulnerability. For example, frequency anomalies for BOPC depend on the number of memory addresses that an attacker requires to manipulate. The frequency anomalies also depend on how the attacker manipulates the memory addresses. In an extreme case, an attacker may require to manipulate only a few memory addresses. Additionally, some vulnerabilities (*e.g.*, `printf()` format string vulnerability) allow direct memory manipulation using a just one command. Thus, the trigger of two or three commands can be hard to detect. This is why the frequency anomaly for BOP may not be as observable as DOP.

V. Conclusion and Future Research Opportunities

In this SoK work, we systematized the current knowledge on data-oriented exploits and applicable defense mechanisms. We experimentally explored the possible side-effects of data-oriented attacks on control-flow behaviors in multiple dimensions. 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 simple investigations in Section IV showed that DOP clearly exhibits some anomalies in the control-flow behavior. Our empirical study using the FlowStitch benchmarks [6] revealed that on average 43% data-oriented gadgets are involved in at least one conditional branch. 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.* [18] 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 IV. 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 involve control flows in multiple locations that are far apart. How to detect incompatible control-flow paths, given a relatively long control-flow sequence, is challenging. Exploring deep learning techniques, such as Long Short-Term Memory (LSTM), may be promising, as LSTM keeps track of temporally distant events.

Selection of Tracing Checkpoints. Due to the storage constraint, it is probably impractical to monitor the complete control-flow transfers of a program. Given a limited overhead budget, how to systematically determine strategic checkpoints for tracing (*e.g.*, setting filters to monitor key functions) would be useful in practice.

Acknowledgement

This work has been in part supported by the Office of Naval Research under Grant ONR-N00014-17-1-2498, National Science Foundation under Grants OAC-1541105 and CNS-1801534, Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), and the Academy of Finland under Grant 309994 (SELIoT). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

[1] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2008.

[2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 552–561, 2007.

[3] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Info. & System Security*, vol. 15, Mar. 2012.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 48–62, 2013.

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

[6] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *USENIX Conference on Security Symposium*, pp. 177–192, 2015.

[7] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 969–986, 2016.

[8] M. Morton, J. Werner, P. Kintis, K. Z. Snow, M. Antonakakis, M. Polychronakis, and F. Monrose, "Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Conference on Security Symposium*, pp. 161–176, 2015.

[10] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Conference on Security Symposium*, 1998.

[11] "Microsoft. Data Execution Prevention (DEP)." http://support.microsoft.com/kb/875352/EN-US/. [Accessed 08-12-2019].

[12] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *ACM Conference on Computer and Communications Security (CCS)*, pp. 298–307, 2004.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2005.

[14] "Microsoft. Return Flow Guard (RGF)." https://technet.microsoft.com/en-us/security/dn425049.aspx. [Accessed 08-12-2019].

[15] "Control-flow Enforcement Technology Preview." https://software.intel.com/sites/default/files/ managed/4d/2a/control-flow-enforcement-technology-preview.pdf. [Accessed 08-12-2019].

[16] "Intel's Memory Protection Extensions." https://software.intel.com/en-us/isa-extensions/intel-mpx. [Accessed 08-12-2019].

[17] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys*, vol. 50, pp. 1–33, Apr. 2017.

[18] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1868–1882, 2018.

[19] A. Baliga, P. Kamat, and L. Iftode, "Lurking in the shadows: Identifying systemic threats to kernel data," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 246–251, 2007.

[20] J. Xiao, H. Huang, and H. Wang, "Kernel data attack is a realistic security threat," in *SecureComm* (B. Thuraisingham, X. Wang, and V. Yegneswaran, eds.), pp. 135–154, 2015.

[21] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn, "Modular protections against non-control data attacks," in *IEEE Computer Security Foundations Symposium*, pp. 131–145, 2011.

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

[23] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, "Enforcing Kernel Security Invariants with Data Flow Integrity," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[24] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 276–291, 2014.

[25] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for Security," *ArXiv e-prints*, June 2018.

[26] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuardTM: Protecting pointers from buffer overflow vulnerabilities," in *USENIX Security Symposium*, vol. 91, 2003.

[27] S. H. Yong and S. Horwitz, "Protecting c programs from attacks via invalid pointer dereferences," in *ACM SIGSOFT Software Engineering Notes*, vol. 28, pp. 307–316, ACM, 2003.

[28] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 263–277, IEEE, 2008.

[29] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "HardBound: architectural support for spatial safety of the c programming language," in *ACM SIGARCH Computer Architecture News*, vol. 36, pp. 103–114, ACM, 2008.

[30] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[31] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIXATC 12)*, pp. 309–318, 2012.

[32] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 147–163, 2014.

[33] C. W. Otterstad, "A brief evaluation of intel® mpx," in *2015 Annual IEEE Systems Conference (SysCon) Proceedings*, pp. 1–7, IEEE, 2015.

[34] G. J. Duck and R. H. Yap, "Heap bounds protection with low fat pointers," in *Proceedings of the 25th International Conference on Compiler Construction*, pp. 132–142, ACM, 2016.

[35] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers.," in *NDSS*, 2017.

[36] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXbounds: Memory safety for shielded execution," in *Proceedings of the Twelfth European Conference on Computer Systems*, pp. 205–221, ACM, 2017.

[37] "Qualcomm Technologies Inc., "Pointer Authentication on ARMv8.3", 2017." https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf. [Online; accessed 03-31-2019].

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

[39] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, pp. 203–216, ACM, 1994.

[40] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 75–88, USENIX Association, 2006.

[41] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 45–58, ACM, 2009.

[42] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with api integrity and multi-principal modules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 115–128, ACM, 2011.

[43] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 457–468, IEEE, 2014.

[44] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pp. 339–348, IEEE, 2006.

[45] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?," in *2012 IEEE Symposium on Security and Privacy*, pp. 571–585, IEEE, 2012.

[46] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 157–168, ACM, 2012.

[47] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–11, IEEE Computer Society, 2013.

[48] P. Larsen, S. Brunthaler, and M. Franz, "Security through diversity: Are we there yet?," *IEEE Security & Privacy*, vol. 12, no. 2, pp. 28–35, 2014.

[49] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, and A.-R. Sadeghi, "Selfrando: Securing the tor browser against de-anonymization exploits," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 454–469, 2016.

[50] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 461–477, IEEE, 2018.

[51] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 268–279, ACM, 2015.

[52] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 367–382, 2016.

[53] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the sixth ACM conference on data and application security and privacy*, pp. 50–61, ACM, 2016.

[54] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make ASLR win the clone wars: Runtime re-randomization.," in *NDSS*, 2016.

[55] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "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*, pp. 1342–1353, ACM, 2014.

[56] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pp. 325–336, ACM, 2015.

[57] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *2015 IEEE Symposium on Security and Privacy*, pp. 763–780, IEEE, 2015.

[58] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 256–267, ACM, 2015.

[59] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 35–46, ACM, 2016.

[60] S. C. Cowan, S. R. Arnold, S. M. Beattie, and P. M. Wagle, "PointGuard: method and system for protecting programs against pointer corruption attacks," July 6 2010. US Patent 7,752,459.

[61] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing.," in *USENIX Security Symposium*, pp. 433–447, 2014.

[62] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 280–291, ACM, 2015.

[63] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, pp. 340–353, ACM, 2005.

[64] I. Fratrić, "ROPGuard: Runtime prevention of return-oriented programming attacks," tech. rep., Technical report, 2012.

[65] M. Zhang and R. Sekar, "Control flow integrity for cots binaries.," in *USENIX Security Symposium*, pp. 337–352, 2013.

[66] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 559–573, IEEE, 2013.

[67] Y. Xia, Y. Liu, H. Chen, and B. Zang, "CFIMon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pp. 1–12, IEEE, 2012.

[68] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch Regulation: low-overhead protection from code reuse attacks," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 94–105, IEEE Computer Society, 2012.

[69] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced control-flow integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pp. 38–49, 2016.

[70] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: hardware-assisted flow integrity extension," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 74, ACM, 2015.

[71] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, November 1996.

[72] D. Litchfield, "Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server," 2003.

[73] "Microsoft Structured Exception Handling Overwrite Protection (SEHOP)." https://support.microsoft.com/en-us/help/956607/how-to-enable-structured-exception-handling-overwrite-protection-sehop/. [Accessed 08-12-2019].

[74] A. Peslyak, ""return-to-libc" attack," *Bugtraq, Aug*, 1997.

[75] H. Shacham, "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*, pp. 552–561, ACM, 2007.

[76] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 575–589, IEEE, 2014.

[77] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 30–40, ACM, 2011.

[78] P. Team, "Pax address space layout randomization (aslr)," 2003.

[79] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, pp. 1–8, ACM, 2009.

[80] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, "CAIN: Silently breaking ASLR in the cloud," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[81] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking holes in information hiding," in *25th USENIX Security Symposium (USENIX Security 16)*, pp. 121–138, 2016.

[82] J. Seibert, H. Okhravi, and E. Söderström, "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*, pp. 54–65, ACM, 2014.

[83] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz, *et al.*, "Address-oblivious code reuse: On the effectiveness of leakage resilient diversity," in *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[84] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*, pp. 574–588, IEEE, 2013.

[85] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 227–242, IEEE, 2014.

[86] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity.," in *USENIX Security Symposium*, pp. 161–176, 2015.

[87] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses.," in *USENIX Security Symposium*, pp. 385–399, 2014.

[88] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans-*

actions on Information and System Security (TISSEC), vol. 13, no. 1, p. 4, 2009.

[89] B. Zeng, G. Tan, and G. Morrisett, “Combining control-flow integrity and static analysis for efficient and validated data sandboxing,” in *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 29–40, ACM, 2011.

[90] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh, “Cryptographically enforced control flow integrity,” *arXiv preprint arXiv:1408.1451*, 2014.

[91] T. H. Dang, P. Maniatis, and D. Wagner, “The performance cost of shadow stacks and stack canaries,” in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 555–566, ACM, 2015.

[92] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Transparent ROP exploit mitigation using indirect branch tracing,” in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pp. 447–462, 2013.

[93] M. Budiu, Ú. Erlingsson, and M. Abadi, “Architectural support for software-based protection,” in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp. 42–51, ACM, 2006.

[94] L. Davi, P. Koeberl, and A.-R. Sadeghi, “Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation,” in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2014.

[95] D. Sullivan, O. Arias, L. Davi, P. Larsen, A.-R. Sadeghi, and Y. Jin, “Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity,” in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2016.

[96] P. Qiu, Y. Lyu, J. Zhang, D. Wang, and G. Qu, “Control flow integrity based on lightweight encryption architecture,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 7, pp. 1358–1369, 2018.

[97] M. Frantzen and M. Shuey, “StackGhost: Hardware facilitated stack protection.,” in *USENIX Security Symposium*, vol. 112, 2001.

[98] “Intel Control-flow Enforcement Technology, Intel Corporat., SantaClara, CA, USA, 2017.” https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. [Online; accessed 03-31-2019].

[99] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, “The dynamics of innocent flesh on the bone: Code reuse ten years later,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1675–1689, ACM, 2017.

[100] “ProFTPD remote exploit.” http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5815. [Online; accessed 04-05-2019].

[101] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[102] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: Exploit hardening made easy.,” in *USENIX Security Symposium*, pp. 25–41, 2011.

[103] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang, “"the web/local" boundary is fuzzy: A security study of chrome’s process-based sandboxing,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 791–804, 2016.

[104] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, “Pt-rand: Practical mitigation of data-only attacks against page tables,” in *Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[105] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis, “Revisiting browser security in the modern era: New data-only attacks and defenses,” in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 366–381, 2017.

[106] K. Sinha and S. Sethumadhavan, “Practical memory safety with REST,” in *Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[107] S. Nagarakatte, J. Zhao, M. Martin, Milo, and S. Zdancewic, “SoftBound: Highly compatible and complete spatial memory safety for C,” in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 245–258, 2009.

[108] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic, “Hardbound: Architectural support for spatial safety of the c programming language,” in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 103–114, 2008.

[109] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, “SGXBOUNDS: Memory safety for shielded execution,” in *European Conference on Computer Systems (EuroSys)*, pp. 205–221, 2017.

[110] G. C. Necula, S. McPeak, and W. Weimer, “CCured: Type-safe retrofitting of legacy code,” in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[111] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[112] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 203–216, 1993.

[113] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, “XFI: Software guards for system address spaces,” in *Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 75–88, 2006.

[114] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, “Software fault isolation with api integrity and multi-principal modules,” in *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 115–128, 2011.

[115] S. Bhatkar and R. Sekar, “Data space randomization,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[116] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz, “Readactor: Practical code randomization resilient to memory disclosure,” in *2015 IEEE Symposium on Security and Privacy (S&P)*, pp. 763–780, 2015.

[117] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, “Data randomization,” Tech. Rep. MSR-TR-2008-120, Microsoft Research, September 2008.

[118] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, “Enhanced operating system security through efficient and fine-grained address space randomization,” in *USENIX Conference on Security Symposium*, pp. 475–490, 2012.

[119] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek, *et al.*, “Hardware assisted randomization of data,” in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 337–358, Springer, 2018.

[120] Q. Chen, A. M. Azab, G. Ganesh, and P. Ning, “Privwatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks,” in *ACM on Asia Conference on Computer and Communications Security*, ASIA CCS ’17, pp. 167–178, 2017.

[121] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, “HDFI: Hardware-assisted data-flow isolation,” in *IEEE Symposium on Security and Privacy (S&P)*, pp. 1–17, 2016.

[122] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[123] Z. Sun, B. Feng, L. Lu, and S. Jha, “OEI: operation execution integrity for embedded devices,” *CoRR*, vol. abs/1802.03462, 2018.

[124] G. Torres and C. Liu, “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*, p. 2, ACM, 2016.

[125] C. Liu, Z. Yang, Z. Blasingame, G. Torres, and J. Bruska, “Detecting data exploits using low-level hardware information: A short time series approach,” in *Proceedings of the First Workshop on Radical and Experiential Security*, pp. 41–47, ACM, 2018.

[126] “The heartbleed bug.”

[127] “Anonymous FTP connections dataset at the Lawrence Berkeley National Laboratory.” https://ee.lbl.gov/anonymized-traces.html. [Online; Accessed 08-12-2019].

[128] D. Pelleg and A. W. Moore, “X-means: Extending k-means with efficient estimation of the number of clusters,” in *International Conference on Machine Learning (ICML)*, 2000.

[129] X. Shu, D. Yao, and N. Ramakrishnan, “Unearthing stealthy program attacks buried in extremely long execution paths,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.