2016 IEEE Symposium on Security and Privacy

Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks

Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, Zhenkai Liang
Department of Computer Science, National University of Singapore
{huhong, shweta24, sendroiu, chuazl, prateeks, liangzk}@comp.nus.edu.sg

***Abstract*—As control-flow hijacking defenses gain adoption, it is important to understand the remaining capabilities of adversaries via memory exploits. Non-control data exploits are used to mount information leakage attacks or privilege escalation attacks program memory. Compared to control-flow hijacking attacks, such non-control data exploits have limited expressiveness; however, the question is: what is the real expressive power of non-control data attacks? In this paper we show that such attacks are Turing-complete. We present a systematic technique called *data-oriented programming* (*DOP*) to construct expressive non-control data exploits for arbitrary x86 programs. In the experimental evaluation using 9 programs, we identified 7518 *data-oriented x86 gadgets* and 5052 *gadget dispatchers*, which are the building blocks for DOP. 8 out of 9 real-world programs have gadgets to simulate arbitrary computations and 2 of them are confirmed to be able to build Turing-complete attacks. We build 3 end-to-end attacks to bypass randomization defenses without leaking addresses, to run a network bot which takes commands from the attacker, and to alter the memory permissions. All the attacks work in the presence of ASLR and DEP, demonstrating how the expressiveness offered by DOP significantly empowers the attacker.**

I. Introduction

Control-hijacking attacks are the predominant category of memory exploits today. The early generation of control-hijacking attacks focused on code injection, while in recent years advanced code-reuse attacks, such as return-oriented programming (ROP) and its variants, have surfaced [1]–[5]. In response, numerous principled defenses for control-hijacking attacks have been proposed. Examples of these include control-flow integrity (CFI) [6]–[11], protection of code pointers (CCFI, CPI) [12], [13], timely-randomization of code pointers (TASR) [14], memory randomization [15], and write-xor-execute (W⊕X, or data-execution prevention, DEP) [16]. All of these defenses aim to ensure that the control flow of the program remains legitimate (with high probability) under all inputs.

A natural question is to analyze the limits of protection offered by control-flow defenses, and the remaining capabilities of the adversary. In a concrete execution, the program memory can be conceptually split into the *control plane* and the *data plane*. The control plane consists of memory variables which are used directly in control-flow transfer instructions (e.g., returns, indirect calls, and so on). In concept, control-flow defenses aim to ensure that the execution of the program stays legitimate — often by protecting the integrity of the control plane memory [12], [14] or by directly checking the targets of control transfers [6]–[10], [17], [18]. However, the data plane, which consists of memory variables not directly used in control-flow transfer instructions, offers an additional source of advantage for attackers. Attacks targeting the data plane, which are referred to as *non-control data attacks* [19], are known to cause significant damage — such as leakage of secret keys (HeartBleed) [20], enabling untrusted code import in browsers [21], and privilege escalation in servers [22]. However, non-control data attacks provide limited expressiveness in attack payloads (e.g., allowing corruption or leakage of a few security-critical data bytes).

In this paper, we show that non-control data attacks with rich expressiveness can be crafted using systematic construction techniques. We demonstrate that non-control data attacks resulting from a single memory error can be Turing-complete. The key idea in our construction is to find *data-oriented gadgets* — short sequences of instructions in the program's control-abiding execution that enable specific operations simulating a Turing machine (e.g., assignment, arithmetic, and conditional decisions). Then, we find *gadget dispatchers* which are fragments of logic that chain together disjoint gadgets in an arbitrary sequence. Such expressive attacks allow the remote adversary to force the program to do its bidding, carrying out computation of the adversary's choice on the program memory. Our constructions are analogous to return-oriented programming, wherein return-oriented instruction sequences are chained [1]. ROP attacks are known to be Turing-complete because of a similar systematic construction [1], [23]. Thus, our attacks enable *data-oriented programming* (*DOP*), which only uses data plane values for malicious purposes, while maintaining complete integrity of the control plane.

Experimental Findings. To estimate the practicality of DOP attacks, we automate the procedure for finding data-oriented gadgets in a tool for Linux x86 binaries. In our evaluation of 9 programs, we statically find 7518 data-oriented gadgets in benign executions of these programs. 1273 of these are confirmed to be reachable from known proof of concept exploits for known CVEs. Gadgets offer a variety of computation controls, such as arithmetic, logical, bit-wise, conditional and assignment operations between values under attacker's influence. Chaining of such gadgets is possible with memory errors if we find dispatchers. We automate the finding of dispatcher loops, such that the vulnerabilities could be used to corrupt the control variable. This allows the attacker to create infinite (or attacker-controlled) repetition. We find 5052

DOI 10.1109/SP.2016.62

of such dispatcher loops in x86 applications. To determine the final feasibility of chaining gadgets using dispatchers (which is a search problem with a prohibitively large space), we resorted to constructing proof-of-concept exploits semi-manually guided by intuition. We show 3 end-to-end exploits in our case-studies. All of our exploits leave the control-plane data unchanged, including all code pointers, and control-flow execution always conforms to the static control-flow graph (CFG). Further, our exploits execute reliably with commodity ASLR and DEP implementations turned on.

Implications. In our first end-to-end exploit, we show how DOP attacks result in bypassing ASLR defenses without leaking addresses to the network. High expressiveness in DOP attacks also allows the adversary to interact repeatedly with the program memory, acting out arbitrary functionality in each invocation. Our second exploit uses the interaction to simulate an adaptive adversary with arbitrary computation power running inside the program's memory space (e.g., a bot on the victim server). We probe the application over 700 times to effect the final attack! Finally, we discuss how to use DOP to subvert several CFI defenses which trust the secrecy or integrity of the security metadata in memory. Specifically, our third exploit changes the permissions of read-only pages to bypass a specific implementation of CFI. As a consequence, we recommend future purely control-flow defenses to consider an adversary model with arbitrary computation and access to memory at the point of vulnerability.

Contributions. In summary, we make the following contributions in this paper:

- *DOP.* We propose data-oriented programming (DOP), a general method to build Turing-complete non-control data attacks against vulnerable programs. We propose concrete methods to identify data-oriented gadgets, gadget dispatchers and a search strategy to stitch these gadgets.
- *Prevalence.* Our evaluation of 9 real world applications shows that programs do have a large number (1273) of data-oriented gadgets reachable from real-world vulnerabilities, which are required by data-oriented programming operations.
- *Practicality.* We show that Turing-complete non-control data exploits for common memory errors are practical. 8 out of 9 applications provide data-oriented gadgets to build Turing-complete attacks. We build 3 end-to-end non-control data exploits which work even in the presence of DEP and ASLR, demonstrating the effectiveness of data-oriented programming.

Our attacks and tools are available at *http://huhong-nus.github.io/advanced-DOP/*.

II. Problem

A. Background: Non-control Data Attacks

Non-control data attacks tamper with or leak security-sensitive memory, which is not directly used in control transfer instructions. Such attacks were conceptually introduced a decade ago by Chen *et al.* to show that they can have serious implications [19]. Recently, Hu *et al.* provided a general construction for automatically synthesizing simple payloads to effect such attacks [22]. Their construction shows that two existing dataflows in the program can be stitched automatically, therefore alleviating the effort for human analysis. The constructed payload required the corruption of a small number (up-to 2 or 3) of non-control-pointers. The attack payloads, however, exhibit limited expressiveness, such as writing a target variable of choice or leaking contents of a sensitive memory region. Such simple payloads can enable privilege escalation and sensitive data-leakage attacks — for instance, Code 1 shows a well-known attack that escalates the program's privileges to root by corrupting one variable (`pw_uid`).

```
1  FILE * getdatasock( ... ) { ...
2    seteuid(0);
3    setsockopt( ... ); ...
4    seteuid(pw->pw_uid);  // corrupted uid (0)
5    ...
6  }
```

Code 1. Code snippet from `wu-ftpd` to demonstrate non-control data attacks. Attackers change `pw->pw_uid` to 0 before the 2nd `seteuid` call to get the root user's privilege.

Is corruption of a few bytes of memory sufficient to enable Turing-complete attacks for remote adversaries? In some programs, the answer is yes. Consider web browsers, which embody interpreters for web languages such as CSS, HTML, JavaScript, and so on. The data consumed by the interpreter is inherently under the remote attacker's control. Further, browsers can import machine code and directly use it, like ActiveX code. By using a few bytes of corruption, it is possible to cause the web browser to making it interpret Turing-complete functionality in another website's origin, or execute arbitrary untrusted code. Such attacks are known in the wild [21], [24]. However, one may argue that such attacks apply only to limited applications such as browsers, which can use process-sandboxing as a second line of defense.

Recently, Carlini *et al.* showed a more subtle example of "interpreter-like" functionality embedded in many common applications [25]. Their work show that certain functions, such as `printf`, take format string arguments and are Turing-complete "interpreters" for the format-string language. Therefore, if a non-control data attack can allow the adversary completely control over the format string argument, then the attacker can construct expressive payloads. However, these examples are specific to certain (4 or 5) functions such as `printf` which permit expressiveness in their format-string language. One way to disable such attacks is to limit the expressiveness of these handful of functions — for instance, the implementation of `printf` in Linux [26] [1] and Windows [27] [2] sanitizes or blocks the use of `%n`, which severely limits the expressiveness of the attack. The question about how expressive are non-control data attacks arising from common memory errors in arbitrary pieces of code is not well-understood. Since non-control data attacks cannot divert

[1] A compile-time flag called FORTIFY_SOURCE enables this check.
[2] "%n" is disabled by default in Visual Studio.

the control flow to arbitrary locations, unlike ROP attacks [1], [23], the expressiveness is believed to be very limited.

B. Example of Data-oriented Programming

In fact, non-control data attacks can offer rich exploits from common vulnerabilities. To see an example, consider the vulnerable code snippet shown in Code 2. The code is modeled after an FTP server, which processes network requests based on the message type. It truncates the "STREAM" message (line 10), maintains the total size of bytes received (line 13) and throttles user requests to a maximum upper limit (line 6). Let us assume that the code has a buffer overflow vulnerability on line 7, failing to check the bounds of the fixed-size buffer `buf` in function `readData`. As a consequence, all local variables, including `srv`, `connect_limit`, `size` and `type` are under the control of attackers.

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

Code 2. Vulnerable FTP server with data-oriented gadgets.

```
1  struct Obj{struct Obj *next; unsigned int prop;}
2  void updateList(struct Obj *list, int addend) {
3     for(; list != NULL; list = list->next)
4        list->prop += addend;
5  }
```

Code 3. A function increments the integer field of a linked list by a given value. It can be simulated by chaining data-oriented gadgets in Code 2.

This code does not invoke any security-critical functions in its benign control-flow, and the vulnerability just corrupts a handful of local variables. Could the adversary exploit this vulnerability to simulate an expressive computation on the program state? A closer inspection reveals that the answer is yes. Consider the individual operations executed by the program. The line 12 is an assignment operation on memory locations pointed by two local variables (`srv` and `type`), which are under the influence of the memory error. Line 10 has a dereference operation, the source pointer (`srv`) for which is corruptible. Similarly, Line 13 has a controllable addition operation. We can think of each of these micro-operations in the program as *data-oriented gadgets*. If we can execute these gadgets on attacker-controlled inputs, and chain their execution in a sequence, then an expressive computation can be executed. Notice that the loop in line 6 to 15 allows chaining and dispatching gadgets in an infinite sequence, since the loop condition is a variable (i.e., `connect_limit`) that is under the memory error's influence. We call such loops *gadget dispatchers*. A sequence of data-oriented gadgets in

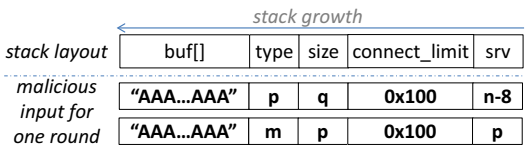

Fig. 1. Malicious input to trigger the loop body in Code 3 by stitching data-oriented gadgets in Code 2. The upper side is the stack layout of Code 2. Refer Table I for details of p, q, m and n.

TABLE I
Simulating the loop body in Code 3 with the data-oriented gadgets in Code 2. In column "Simulated Instr.", highlighted instructions are useful for the simulation, while other instructions are side effects of the attack.

Overflow	Executed Instr. (Code 2)	Simulated Instr. (Code 3)

type ← p
size ← q
srv ← n-8
	if(*type == NONE) break;
	srv->typ = *type;
	srv->total += *size;
		if(list == NULL) break;
		srv = list;
		list->prop += addend;

type ← m
size ← p
srv ← p
	if(*type == NONE) break;
	if(*type == STREAM)
	*size = *(srv->cur_max);
		if(list == NULL) break;
		if(list == STREAM)
		list = list->next;

p – &list; q – &addend; m – &STREAM; n – &srv		

Code 2 would allow the remote adversary to simulate the function shown in Code 3, which maintains a linked list of integers in memory and increments each integer by a desired value. Table I illustrates how the code in the loop body gets simulated with the malicious input in Figure 1. Attackers can repeatedly send the same input sequence to implement the `updateList` function in Code 3.

This non-control data attack shows subtle expressiveness in payloads and prevalence: with a single memory error, it re-interprets the vulnerable server as a virtual CPU, to perform an expressive calculation on behalf of attackers. It does not require any specific security-critical data or functions to enable such attack. The control flow conforms to the precise CFG.

C. Research Questions

In this paper, we aim to answer the following questions about non-control data attacks:

- **Q1**: How often do data-oriented gadgets arise in real-world programs? How often do gadget dispatchers exist?
- **Q2**: Is it possible to chain gadgets for a desired computation? Can attackers build Turing-complete attacks with this method?
- **Q3**: What is the security implication of this attack method for current defense mechanisms?

III. Data-Oriented Programming

We illustrate the idea behind a general technique called Data-Oriented Programming (DOP) that can simulate Turing-complete computations by exploiting a memory error.

A. DOP Overview

Data-oriented programming is a technique that allows the attacker to simulate expressive computations on the program memory, without exhibiting any illegitimate control flow with respect to the program CFG. As shown in Section II-B, the key is to manipulate non-control data such that the executed

TABLE II
MINDOP language. To simulate (conditional) jump, data-oriented gadget changes the virtual input pointer (vpc) accordingly.

Semantics	Instructions in C	Data-Oriented Gadgets in DOP
arithmetic / logical	`a op b`	`*p op *q`
assignment	`a = b`	`*p = *q`
load	`a = *b`	`*p = **q`
store	`*a = b`	`**p = *q`
jump	`goto L`	`vpc = &input`
conditional jump	`if a goto L`	`vpc = &input if *p`

p – &a; q – &b; op – arithmetic / logical operation

instructions do the attacker's bidding. In order to give a concrete and systematic construction, we define a simple mini-language called MINDOP with a virtual instruction set and virtual register operands, in which the attacker's payload can be specified. We show how MINDOP can be simulated by small snippets of x86 instructions that are abundant in common real-world programs on Linux, as our empirical evaluation in Section V confirms. MINDOP is Turing-complete, which we establish in Section III-D.

The MINDOP language (shown in Table II) has 6 kinds of virtual instructions, each operating on virtual register operands. The first four virtual instructions include arithmetic / logical calculation, assignment, load and store operations. The last two virtual operations, namely conditional and unconditional jumps, allow the implementation of control structures in a MINDOP virtual program. Each virtual operation is simulated by real x86 instruction sequences available in the vulnerable program, which we call *data-oriented gadgets*. The control structure allows chaining of gadgets, and the x86 code sequences that simulate the virtual control operations are referred to as *gadget dispatchers*. None of the gadgets or dispatchers modify any code-pointers or violate CFI in the real program execution. We next explain each virtual operation and show concrete real-world gadgets that simulate them.

B. Data-Oriented Gadgets

Virtual operations in MINDOP are simulated using concrete x86 instruction sequences in the vulnerable program execution. Such instruction sequences or gadgets read inputs from and write outputs to memory locations which simulate virtual register operands in MINDOP. Hardware registers are not a judicious choice for simulating virtual registers because the original program frequently uses hardware registers for its own computation. Gadgets are scattered in the program logic, within the legitimate CFG of the program. As a result, between two gadgets there may be several uninteresting instructions which may clobber hardware registers and the memory state outside of the attacker's control. Therefore, in MINDOP we implement virtual registers with carefully-chosen memory locations (not hardware registers) used only under the control of gadget operations.

Conceptually, a data-oriented gadget simulates three logical micro-operations: the load micro-operation, the intended virtual operation's semantics, and the final store micro-operation. The load micro-operation simulates the read of the virtual register operand(s) from memory. The store micro-operation writes the computation result back to a virtual register. The operation's semantics are different for each gadget. A number of different x86 instruction sequences can suffice to simulate a virtual operation. The x86 instruction set supports several memory addressing modes, and as long as the order of the micro-operations is correct, different sequences can work. As a concise example, the x86 instruction `add %eax, (%ecx)` performs all three micro-operations (load, arithmetic and store) in a single x86 instruction. We later provide other gadget implementations as well.

TABLE III
Example data-oriented gadget of addition operation. The first row is the C code of the gadget, and the second row is the corresponding assembly code.

C Code	`srv->total += *size;`
ASM Code	1 `mov (%esi), %ebx` **`//load micro-op`** 2 `mov 0x4(%edi), %eax` **`//load micro-op`** 3 `add %ebx, %eax` **`//addition`** 4 `mov %eax, 0x4(%edi)` **`//store micro-op`**

Data-oriented gadgets are similar to code gadgets employed in return-oriented programming (ROP) [1], or in jump-oriented programming (JOP) [2]. They are short instruction sequences and are connected sequentially to achieve the desired functionality. However, there are two differences between data-oriented gadgets and code gadgets. First, data-oriented gadgets require to deliver operation result with memory. In contrast, code gadgets can use either memory or register to persist outputs of a gadget. Second, data-oriented gadgets must execute in at least one legitimate control flow, and need not execute immediately one after another. In fact, they can be spread across several basic blocks or even functions. In contrast, code gadgets need not execute in any benign control-flow path of the program, and may even start at invalid instruction boundaries. Data-oriented gadgets have more stringent requirements than code gadgets in general. However, we show that such gadgets exist with examples from real-world applications.

Simulating Arithmetic Operations. Addition and subtraction can be simulated using a variety of x86 instructions sequences that we find empirically. Table III shows one example of addition gadget with C and the assembly representation. This gadget is modeled from the real-world program ProFTPD [28]. In the assembly representation, the code in line 1 and line 2 constitute the load micro-operation. The code in line 3 implements the addition, and line 4 is the store micro-operation.

With addition over arbitrary values, it is possible to simulate multiplication efficiently if the language supports conditional jumps. MINDOP supports conditional jumps which allow to check if a value is smaller / greater than a constant. To see why this combination is powerful, note that we can compute the bit-decomposition of a finite-size integer. To compute the most significant bit of `a`, we can add `a` to itself (equivalently left-shifting it) and conditionally jump based on the carry bit. Proceeding by repetition, we can obtain the bit-decomposition of `a`. With bit-decomposition, simulating a multiplication `a·b` reduces to the efficient shift-and-add procedure, adding `a` to itself in each step conditioned on the bits in `b`. Converting a

TABLE IV
Example data-oriented gadget of assignment operation.

C Code	`srv->typ = *type;`
ASM Code	`1 mov (%esi), %ebx` **`// load micro-op`** `2 mov %ebx, %eax` **`// move`** `3 mov %eax, 0x8(%edi)` **`// store micro-op`**

TABLE V
Example data-oriented gadget of dereference operation. The first row gives three examples. The second row shows the assembly code of the first one.

C Code	`LOAD1: *size = *(srv->cur_max);`
	`LOAD2: memcpy(dst, *src_p, size);`
	`STORE: memcpy(*dst_p, src, size);`

ASM Code	(of `LOAD1`) `1 mov (%esi), %ebx` **`// load micro-op`** `2 mov (%ebx), %eax` **`// load`** `3 mov %eax, (%edi)` **`// store micro-op`**

bit-decomposed value to its integer representation is similarly a multiply-and-add operation over powers of two. Bit-wise operations are simply arithmetic on the bit-decomposed versions.

Simulating Assignment Operations. In MINDOP, assignment gadgets read data from one memory location and directly write to another memory location. In this case, we can skip the load section of the destination operand. The C code and ASM code of an assignment gadget is shown in Table IV.

Simulating Dereference (Load / Store) Operations. The load and store instructions in C require memory dereferences, which take one register as address and visits the memory location for reading or writing. In data-oriented programming, registers are simulated by memory, therefore the memory dereference is simulated by two memory dereferences: the first memory dereference to simulate the register access, and a second memory dereference with the first dereference result (the register value) as the address. As shown in Table II, the memory dereference `*b` in the load instruction in C is represented by `**q`, where `q` is the address of `b`, `*q` is the value of `b`, and `**q` is the final memory value `*b`. A similar representation is used in the store gadget. We show two examples of load gadgets and one example of a store gadget in Table V in C representation, and the assembly representation of the first load gadget. As we can see from the assembly code, there are still three sections in load / store gadgets, with the semantics on memory dereference with loaded operands.

C. Gadget Dispatcher

Various gadgets can be chained sequentially by gadget dispatchers to enable recursive computation. Gadget dispatchers are sequences of x86 instructions that equip attackers with the ability to repeat gadget invocations and, for each invocation, to selectively activate specific gadgets. One common sequence of x86 instructions that can simulate gadget dispatchers is a loop, which iterates over computation that simulates gadgets and should have a selector in it. Each iteration executes a subset of gadgets using outputs from gadgets in the previous iteration. To direct the outputs of one gadget in iteration i into the inputs to a gadget in iteration $i+1$, the selector changes the load address of iteration $i+1$ to the store addresses of iteration i. The selector's behavior is controlled by attackers through the memory error. In our running example in Code 2, line 6 and

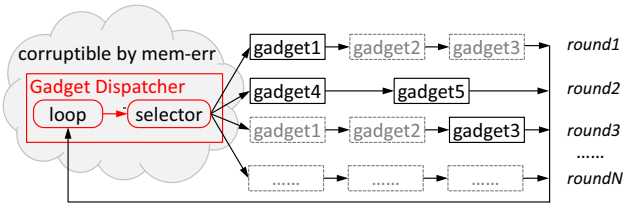

Fig. 2. The design model of DOP in MINDOP. The gadget dispatcher includes a loop and a selector. The loop keeps the program passing by the selector and various data-oriented gadgets. For each round, the selector is controlled by the memory error to activate particular data-oriented gadgets.

7 in the loop constitute a dispatcher. The selector on line 7 is the memory error itself, which repeatedly corrupts the local variables to setup the execution of gadgets in that iteration. The corruption is done in a way that it enables only the gadgets of the attacker's choice. These gadgets take as input the outputs of the previous round's gadget by selectively corrupting operand pointers. The remaining gadgets may still get executed, but their inputs and outputs are set up such that they behave like NOPs (operating on unused memory locations). Figure 2 shows the design model of data-oriented programming in MINDOP. The left part is the gadget dispatcher inside the vulnerable program, which is corruptible by the memory error; the solid gadgets are activated in iteration i and the gray gadgets are executed like NOPs.

It remains to explain in iteration i, how to selectively activate a particular gadget in that iteration and whether the simulation should continue to iteration $i+1$. Our running example in Code 2 shows a scenario where the attacker can "interact" with the program by repeatedly corrupting program variables at line 7 using a buffer overflow. This attack is an *interactive* attack, where the attacker can prepare the memory state at the start of loop iteration i in a way that the desired gadget works as required and other gadgets operate on unused memory. Let M_j be the state of memory for executing gadget j selectively. In an interactive attack, the attacker can corrupt local variables to configure M_j to execute j in that round, and provide multiple rounds of such malicious inputs to perform an expressive computation. When the attacker wishes to terminate the loop, it can corrupt the loop condition variable to stop.

Another class of DOP attacks are *non-interactive*, whereby the attacker provides the entire malicious input as a single data transmission. In such a scenario, all the memory setup and conditions for deciding loop termination and selective gadget activation need to be encoded in a single malicious payload. To support such attacks, MINDOP has two virtual operations that enable conditional chaining of operations, or virtual jumps. The basic idea is as follows: the attacker provides the memory configuration M_j necessary for each gadget j to be selectively executed in a particular iteration in the input payload. In addition, it keeps a pointer called the *virtual PC* which points to the desired configuration M_j at the start of each iteration. It suffices to corrupt only the virtual PC, so that the program execution in that iteration operates on the configuration M_j. To decide how to switch to

M_k in the next iteration, MINDOP provides virtual operations that set the virtual PC, conditionally or unconditionally. The dispatcher loop can be conditionally terminated by using a specific memory configuration M_{exit} which sets the loop condition variable appropriately. We provide example gadgets that simulate such virtual operations below.

Simulating Jump Operations. The key here is to identify a suitable variable to implement a virtual PC which can be corrupted in each loop iteration. One example of such a gadget is the Code 4 taken from the real ProFTPD program [28][3]. There is a memory pointer `pbuf->current` that points to the buffer of malicious network input. In each loop iteration, the code reads one line from the buffer and processes it in the loop body — thus the pointer is a useful candidate to simulate a virtual PC. To simulate an unconditional jump, attackers just prepare the memory configuration to trigger another operation gadget (e.g., addition, assignment) to change the value of the virtual PC. For example, if attackers want the MINDOP program to jump from operation i to j, they just need to prepare the memory configuration M_k after M_i, so that the operation k will change the virtual PC to point to M_j. Furthermore, there are two ways to simulate a conditional jump. One case is that reading the memory configuration with virtual PC is conditional. Attackers just use operation k to set the proper variable as the reading condition. Another case is that the operation k's execution conditionally depends on a data variable.

```
1  void cmd_loop(server_rec *server, conn_t *c) {
2    while (TRUE) {
3      pr_netio_telnet_gets(buf, ..);
4      cmd = make_ftp_cmd(buf, ...);
5      pr_cmd_dispatch(cmd); // dispatcher
6    }
7  }
8  char *pr_netio_telnet_gets(char * buf, ...) {
9    while(*pbuf->current!='\n' && toread>0)
10     *buf++ = *pbuf->current++;
11 }
```

Code 4. Example data-oriented gadget of jump operation.

The virtual PC in non-interactive mode requires a dedicated space for malicious input and a controllable input pointer. In Section V we show the details of the identified virtual PCs in real-world programs. Note that interactive attack model does not require a virtual PC as attackers can dynamically decide the next gadget based on the network message received from the victim program in each iteration.

D. MINDOP *is Turing-Complete*

To show that MINDOP is Turing-complete, we show how the classical construction of a Turing machine can be simulated in MINDOP. A Turing machine M is a tuple $(Q, q_0, \Sigma, \sigma_0, \delta)$ where,

- Q is a finite set of states,
- q_0 is a distinguished start state such that $q_0 \in Q$
- Σ is a finite set of symbols
- σ_0 is a distinguished blank symbol such that $\sigma_0 \in \Sigma$
- δ is a transition table mapping a partial function $Q \times \Sigma \mapsto \Sigma \times \{L, R\} \times Q$

Representation. In the context of DOP, we set-up the following data structures in the victim program's memory to represent our Turing Machine: A q_{cur} to hold the current state, where q_{cur} is a member of set of all possible states (Q). A pointer $tape_{head}$ to track the cell on the tape containing the current symbol S_{cur}, where S_{cur} is a member of set of all possible symbols (Σ). Note that since the tape is linear, $tape_{head} - 1$ points to left part of the tape w.r.t. current position, and $tape_{head} + 1$ points to the right part to the tape. A pointer TT_{base} to access a two-dimensional array that stores the transition table. The transition table uses the current state q_{cur} and the current symbol S_{cur} as indexes. Pointers q_{next}, S_{next}, D hold the next state, next symbol and the movement direction (left or right) respectively.

Simulating Steps of A Turing Machine. In the first step of the attack, we invoke the memory gadgets to load the input and transition table into the program memory. We also initialize q_{cur} to q_0 and $tape_{head}$ to S_{cur}. For achieving this, the attacker crafts a payload which will execute the sequence of operations shown in Listing 5. This requires three basic types of gadgets: assignment (`MOV`), dereference (`LOAD` and `STORE`) and addition (`ADD`).

```
1  MOV tape_head, temp
2  STORE input_0, temp   ;start writing input
3  ADD temp, 1
4  STORE input_1, temp
5  ....
6  STORE input_n, temp   ;end writing input
7  LOAD tape_head,S_0    ;init S_0
8  MOV q_0, q_cur        ;init q_cur
9  MOV TT_base, address  ;start writing trans tab
10 MOV temp, TT_base
11 STORE value_0, temp
12 ADD temp, 1
13 STORE value_1, temp
14 ADD temp, 1
15 ....
16 STORE value_n, temp   ;end loading trans tab
```

Code 5. Data-oriented gadget sequence to initialize the Turing Machine.

Once the attacker sets up the Turing machine, the next aim is to execute the machine with the provided input on the tape. The classical Turing machine step comprises of four sub-steps: (a) read the current tape symbol (b) use the symbol and the state to consult the transition table and get the next state and symbol (c) write the new symbol to the tape and update the state (d) move the tape head to left or right. Listing 6 shows the sequence of gadgets that should be chained together to simulate such a step in the attacker's Turing machine. Note that it is a fixed chain of gadgets only comprising of assignment, dereference and addition operations.

Accessing Transition Tables. Each step in the machine consults the transition table by using the current state and the current tape symbol. We place our transition table in the memory in such a way that the comparison operation to search the transition table is folded into a direct lookup in a two-

[3]Although ProFTPD provides an interactive attack mode, it also allows non-interactive attack with this jump gadget.

dimensional array. Specifically, we use the addition gadget to first calculate the offset in the transition table. This calculation is done dynamically based on the current symbol and state. Once we obtain the offset, we use it to lookup the next state and symbol. Next, we update the current state, write the new symbol, and move the tape head (See Appendix A for an example). The attacker aims to carry out all the above sub-steps repeatedly until the Turing machine reaches the final or halt state. When the machine does reach a halt state, the lookup-table can encode a specific output symbol to terminate. For instance, the output symbol could terminate the dispatcher loop to proceed with the original program execution.

```
1  LOAD vptr, s_cur         ;read from tape
2  MOV TT_base, temp
3  ADD temp, q_cur          ;get the row
4  LOAD temp, temp
5  ADD temp, s_cur          ;get the column
6  LOAD temp, TT
7  LOAD TT, q_cur           ;set the new state
8  ADD TT, 1
9  LOAD TT, s_cur
10 ADD TT, 1
11 LOAD TT, D
12 STORE s_cur, tape_head   ;write to tape
13 ADD tape_head, D         ;move the head
14 MOV loop_counter, temp
15 ADD temp, 16
16 MOV temp, loop_counter
```

Code 6. Gadget sequence to simulate one step in the Turing Machine.

Putting It All Together. To cascade multiple Turing machine steps, the attacker has to ensure that the victim program's dispatcher loop does not exit. Line 14-16 in Code 6 show one possible way to achieve this by incrementing the vulnerable program's loop counter variable at the end of every step in the Turing machine. Depending on the nature of the gadget dispatcher in the program, the attacker can chose alternative ways to achieve the same. In order to successfully execute any arbitrary computation in the vulnerable program's memory, the attacker constructs a payload such that it first executes the gadgets for initialization and then keeps pumping the payload to execute machine step gadgets repeatedly until the victim program terminates. Thus, we prove that if the program has three stitchable gadgets for assignment, dereference and addition within a dispatcher loop, then it is possible to mount Turing-complete DOP attacks.

IV. DOP Attack Construction

Constructing DOP attacks against a vulnerable program requires a concrete memory error and specification of the malicious behavior. Our analysis first identifies concrete program gadgets and dispatchers to simulate MINDOP operations, and then we synthesize a malicious input to execute MINDOP operations exploiting an existing concrete memory error.

A. Challenges

Though the concept of data-oriented programming is intuitive, it is challenging to construct data-oriented attacks in real-world programs. Unlike in ROP, where attackers completely harness the control flow, DOP is constrained by the application's original control flow. Following challenges arise in constructing DOP attacks:

- **Data-oriented gadget identification**. To perform arbitrary computations, we need to find data-oriented gadgets to simulate basic MINDOP operations. However, most of the data-oriented gadgets are scattered over a large code base, which makes manual identification difficult. We use static analysis as an aid in identifying these gadgets.
- **Gadget dispatcher identification**. Our gadget dispatcher requires a loop with various gadgets and a selector controlled by the memory error. But it is possible to have the selector and gadgets inside the functions called from the loop body. We should take such cases into consideration to identify all dispatchers.
- **Data-oriented gadget stitching**. The reachability of gadgets depend on concrete memory errors. We need to find malicious input that makes the program execute selected gadgets with the expected addresses and order. Since data-oriented programming corrupts substantial memory locations, we also need to avoid program crashes.

In the rest of this section, we discuss our techniques to address the challenges in identifying data-oriented gadgets, gadget dispatchers and stitching them for real-world attacks.

B. Gadget Identification

A useful data-oriented gadget needs to satisfy the following requirements:

- MINDOP **semantics.** It should have instructions for the load micro-operation, the store micro-operation, and others simulating semantics of MINDOP, as we discuss in Section III-B.
- **Gadget internal order.** The three micro-operations should appear in the load-operation-store order, and this correct order should show up in at least one legitimate control flow.

We perform static data-flow analysis to aid the identification of such data-oriented gadgets and generate a set of over-approximated gadgets verifiable by manual / dynamic analysis. We compile the program source code into LLVM intermediate representation (IR) and perform our analysis on LLVM IR. LLVM IR provides more program semantics than binary while avoiding the parsing of program source code. It also allows language-agnostic analysis of the source code written in any language that has a LLVM frontend. Our analysis iterates through all functions in the program (See Algorithm 1). We treat each store instruction in the function as a store micro-operation of a new potential gadget. Then our analysis uses a backward data-flow analysis to identify the definitions of the operands in the store instruction. The generated data-flow contains the instructions that derive the operands, like loaded from memory or calculated from registers. If there is at least one load operation present in the data-flow, we mark it as a data-oriented gadget.

Gadget Classification. We classify data-oriented gadgets into different categories based on their semantics and computed

Algorithm 1: Data-oriented gadget identification.

```
Input: G:- the vulnerable program
Output: S:- data-oriented gadget set
S = ∅;
FuncSet = getFuncSet(G)
foreach f ∈ FuncSet do
    cfg = getCFG(f)
    for instr = getNextInstr(cfg) do
        if isMemStore(instr) then
            gadget = getBackwardSlice(instr, f)
            input = getInput(gadget)
            if isMemLoad(input) then
                S = S ∪ {gadget}
```

Algorithm 2: Gadget dispatcher identification.

```
Input: G:- the vulnerable program
Output: D:- gadget dispatcher set
D = ∅;
FuncSet = getFuncSet(G)
foreach f ∈ FuncSet do
    foreach loop = getLoop(f) do
        loop.gadgets = ∅
        foreach instr = getNextInstr(loop) do
            if isMemStore(instr) then
                loop.gadgets ∪= getGadget(instr)
            else if isCall(instr) then
                target = getTarget(instr)
                loop.gadgets ∪= getGadget(target)
        if loop.gadgets != ∅ then
            D = D ∪ {loop}
```

variables. Gadgets with the same semantics are functional-equivalent to simulate one MINDOP operation. The assignment gadgets can be used to prepare operands for other gadgets. Conditional gadgets are useful to implement advanced calculations from simple gadgets (like simulating multiplication with conditional addition in Section III-B). There are no function call gadgets in data-oriented programming, as it does not change the control data. Based on the computed variables, we further classify gadgets into three categories: *global gadget*, *function-parameter gadget* and *local gadget*. Global gadgets operate on global variables. Memory errors can change these variables from any location. A function-parameter gadget operates on variables derived from function parameters. Memory errors that can control the function parameters can use gadgets in this category. Local gadgets compute on local variables, where only the memory errors inside the function can activate them. One concrete memory error can use gadgets in various categories. For example, a stack buffer overflow vulnerability can use local gadgets if it can corrupt related local variables. It can also use function-parameter gadgets if the corrupted local variables are used as parameters of function calls. If the buffer overflow can be exploited to achieve arbitrary memory writing, even the global gadgets can be used to build attacks [4].

We use classification to prioritize gadget selection: global gadgets are prioritized over function-parameter gadgets, and local gadgets are considered at last. We further prioritize the identified potential gadgets based on their features, which include the length of the instruction sequence and the number of simulated operations. Shorter instruction sequences with single MINDOP semantic are prioritized over longer, multi-semantic instruction sequences.

C. Dispatcher Identification

We use static analysis on LLVM IR for the initial phase of dispatcher identification. In this step, our method does not consider any specific memory error. Algorithm 2 gives the dispatcher identification algorithm. Since loops are necessary for attackers to repeatedly connect gadgets, we first identify all possible loops in the program. For each loop, we scan the instructions in the loop body to find interesting gadgets with Algorithm 1. For function calls within the loop, we step into functions through the call graph and iterate through all instructions inside. This gives us an over-approximate set of gadget candidates for a particular dispatcher. As with the gadget finding, we also prioritize dispatchers based on loop size and loop condition.

The second phase of dispatcher identification correlates the identified dispatcher candidates with a known memory error. In this phase, we use a static-dynamic approach to provide identification results with varying degrees of coverage and precision. Static analysis provides a result with larger coverage but less precise, while dynamic analysis allows for the converse. In our static analysis, the correlation is done by reachability analysis of loops based on program's static CFG. We mark a loop as reachable if it enfolds the given memory error. For dynamic analysis, we consider the function call trace after the execution of the vulnerable function until the program termination. Any loops inside the called functions are treated to be under the control of memory error. We merge the static analysis result and dynamic analysis result as the final set of gadget dispatchers.

D. Attack Construction

We manually construct our final attacks with data-oriented programming using the results of our previous analysis. For a given concrete memory error, the available gadgets and dispatchers rely on the location of the vulnerable code in the program, while the stitchability of gadgets depends on the corruptibility of the memory error. To connect two disjoint data-oriented gadgets, attackers should have the control over the address in the load micro-operation of the second gadget or the address in the store micro-operation of the first gadget. Attackers can modify the addresses into expected values when the address values are known in advance (through information leakage or deterministic address analysis [22]). Based on the gadget classification, we complete the stitching steps manually, with the following method.

1) **Gadget preparation (Semi-automated)**. Given a memory error, we locate the vulnerable function from the

[4]Like the cases in Section V

TABLE VI

Prevalence of DOP gadgets & dispatchers. Columns 1-3 present the details of the selected 9 programs. Columns 4 denotes the total number of identified dispatchers, while Column 5 represents the number of dispatchers containing at least one gadget. Columns 6-20 report the number of gadgets for each type where, **G** denotes **global** gadgets, **FP** denotes **function-parameter** gadgets, and **H** denotes the operands in the gadgets are **hybrid**.

Vulnerable Application			Dispatchers		Assignment			Load / Store			Arithmetic			Logical			Conditional		
Name	**Version**	**LOC**	**Total**	**Used**	**G**	**FP**	**H**	**G**	**FP**	**H**	**G**	**FP**	**H**	**G**	**FP**	**H**	**G**	**FP**	**H**
bitcoind	0.11.1	455041	88	9	0	2	0	0	0	0	0	50	0	0	40	0	0	6	0
musl libc	1.1.7	84643	2165	768	80	303	126	935	321	76	595	542	54	160	292	17	70	55	17
BusyBox	1.24.1																		
Wireshark	1.8.0	2629412	961	102	33	33	41	49	41	4	37	125	8	3	13	6	3	7	4
nginx	1.4.0	100252	689	204	12	69	54	32	974	40	12	177	26	28	265	7	3	22	6
mcrypt	2.6.8	51673	71	9	65	6	0	0	0	8	13	0	0	21	10	0	9	0	1
sudo	1.8.3	94492	67	16	11	9	0	11	8	5	3	9	2	5	0	2	3	0	0
ProFTPD	1.3.0	202206	586	180	6	23	15	43	81	48	53	23	2	16	13	2	7	2	0
sshd	1.2.27	38236	222	88	3	45	28	6	174	9	10	232	17	6	305	24	3	28	0
WU-FTPD	2.6.0	25968	203	67	40	3	9	34	1	1	114	0	0	5	3	0	3	2	3
		TOTAL	**5052**	**1443**															**7518**

program source code. Then we identify the gadget dispatchers that enfold the vulnerable code and collect the data-oriented gadgets.

2) **Exploit chain construction (Manual)**. We take the expected malicious MINDOP program as input. Each MINDOP operation can be achieved by any data-oriented gadget in the corresponding functional category. We select gadgets based on their priorities.
3) **Stitchability verification (Manual)**. Once we get a chain of data-oriented gadgets for desired functionality, we verify that every stitching is possible with the gadget dispatcher surrounding them. We feed concrete input to the program to trigger memory errors to connect expected gadgets. If the attack does not work, we roll back to Step 2) to select different gadgets and try the stitching again.

V. EVALUATION

In this section, we measure the feasibility of data-oriented programming and answer the research questions outlined in Section II-C. We first show the prevalence of data-oriented gadgets and gadget dispatchers in real-world x86 programs (**Q1**). We sample the identified gadgets and empirically verify if they are stitchable with known CVEs. We find both Turing-complete data-oriented gadgets as well as dispatchers in interactive and non-interactive mode (**Q2**). We demonstrate three end-to-end case-studies which use DOP to exploit the program while bypassing ASLR and DEP to highlight the utility of Turing-completeness (**Q3**).

Selection of Benchmarks. We select 9 widely used applications with publicly known CVEs for our evaluation. These applications provide critical network services (like FTP, HTTP, cryptocurrency) and thus are common targets of real-world exploits. Specifically, we study FTP servers (WU-FTPD [29], ProFTPD [28]), HTTP server (nginx [30]), daemons (bitcoind [31], sshd [32]), network packet analyzer (Wireshark [33]), user library musl libc [5] [36], and common user utilities (mcrypt [37], sudo [38]).

[5]We analyze standard C library musl libc instead of glibc [34] because glibc cannot be compiled with LLVM. In our analysis, we use BusyBox [35] built against musl libc.

A. Feasibility of DOP

We study our 9 applications and measure how many x86 gadgets in these programs can simulate MINDOP operations. We aim to evaluate the following four aspects in our analysis:

- Empirically justify the choice of operations in MINDOP based on the prevalence of x86 gadgets.
- Study the distribution of various types of gadgets based on the scope of input operands.
- Measure the reachability of these x86 gadgets in concrete executions in presence of an exploitable memory error.
- Verify if the memory errors (in the public CVEs) have the capability to control the input operands and activate the gadgets in concrete executions.

Choice of MINDOP **Operations**. Table VI shows our static analysis results, including the number of x86 gadgets and gadget dispatchers to simulate MINDOP operations.

- *x86 Gadgets.* We identified 7518 data-oriented gadgets from 9 programs. 8 programs provide x86 data-oriented gadgets to simulate all MINDOP operations. In fact, there are multiple gadgets for each operation. These gadgets provides the possibility for attackers to enable arbitrary calculations in program memory. Another program, bitcoind, contains x86 gadgets to simulate MINDOP operations except load and store. This result implies that real-world applications do embody MINDOP operations and are fairly rich in DOP expressiveness.
- *x86 Dispatchers.* Our programs contain 5052 number of gadget dispatchers in total, such that each program has more that one dispatcher (See Column 4 in Table VI). 1443 of these dispatchers contain x86 gadgets of our interest (See Column 5 in Table VI). More importantly, programs such as sudo with relatively fewer number of loops w.r.t LOC, still contains 16 dispatchers to trigger x86 gadgets. This means that the dispatchers are abundant in real-world programs to simulate MINDOP operations.

Gadgets Classification. We classify our x86 gadgets into three categories based on the scope of the operands as discussed in Section IV-B. The inputs of global gadgets can be controlled by a memory error from any location. The inputs of function-parameter gadgets can be controlled only if the memory error

TABLE VII

Reachability and corruptibility of x86 gadgets in the presence of a specific memory error. Columns 2-4 present the CVE number, type and capacity of the vulnerability as format string vulnerability (FSV), integer overflow (IO), stack buffer overflow (SBO), arbitrary write (AW) and stack write (SW). Column 5 denotes the number of functions executed after the vulnerable function in the program. Columns 6-7 report the total number of dispatcher loops executed and how many of them execute at least one data gadget respectively. Columns 8-12 represent the number of gadgets in each type executed within these dispatchers and a ✓implies that at least one gadget is confirmed to be stitchable. Column 13 reports the total number of gadgets. Columns 14-15 report if the vulnerability can be used to maintain a virtual PC and build Turing-complete exploit respectively.

Vulnerable Application				Func	Dispatchers		Assign-ment		Load / Store		Arith-metic		Logical	Condi-tional	Total Gadgets	Virtual PC?	Is TC?
Name	CVE	Type	Cap	Exec	Exec	Used											
bitcoind	2015-6031 [39]	SBO	SW	0	0	0	0		0		0		0	0	0		
musl libc + ping	2015-1817 [40]	SBO	AW	83	88	16	18	✓	45		6	✓	19 ✓	0	88		
Wireshark	2014-2299 [41]	SBO	AW	152	44	1	1	✓	1	✓	2	✓	0	1 ✓	5	✓	✓
nginx	2013-2028 [42]	SBO	AW	82	91	30	39		441		68	✓	119	11	678		
mcrypt	2012-4409 [43]	SBO	AW	31	10	2	1	✓	0		5		1	0	7		
sudo	2012-0809 [44]	FSV	AW	27	9	2	3		12		4	✓	2	2	23		
ProFTPD	2006-5815 [45]	SBO	AW	146	92	31	15	✓	69	✓	22	✓	18	2 ✓	126	✓	✓
sshd	2001-0144 [46]	IO	AW	91	34	20	10		11		19	✓	120	0	160		
WU-FTPD	2000-0573 [47]	FSV	AW	23	36	9	47	✓	21	✓	102	✓	8	8	186	✓	
			TOTAL	635	404	110	134		600		228		287	24	**1273**	**3**	**3**

occurs before the parent function. We currently ignore the local gadgets as it requires the memory error to occur in the function body before it. Instead, we consider the gadgets taking both global inputs and function-parameter inputs, classified as hybrid gadgets. An arbitrary memory error provides partial control over hybrid gadgets. Table VI reports the number of gadgets in each category for our examples. 8 out of 9 programs have at least one class of gadget for each operation, which shows that highly controllable gadgets are common in real-world vulnerable programs.

Execution Reachability from Memory Errors. The nature of the memory vulnerability — location in the code and the corruptible memory region — decides if the execution can reach a specific gadget or not. To estimate how many gadgets in the program are reachable, we select one concrete vulnerability from the CVE database per program (See Column 2 in Table VII). We run the vulnerable program with the given CVE PoC to get the dynamic function call trace, including the vulnerable function. From the function call trace, we identify the functions invoked by the vulnerable function, and loops surrounding the vulnerable function during the execution. The gadgets inside the invoked functions and enfolding loops are the reachable gadgets from the dispatcher. Table VII shows the number of reachable gadgets in our programs. In total there are 1273 reachable gadgets via the listed CVEs. 4 out of 9 programs have at least one dispatcher and one gadget of each type reachable from the selected CVE, which can be used to simulate all operations in MINDOP. Thus, these selected real-world vulnerabilities have the ability to reach a large number of data-oriented gadgets and invoke many dispatchers.

Corruptibility of the CVE. The fact that a gadget is reachable does not necessarily imply that the attacker can always control the inputs to the gadget. For example, memory errors with only stack access can use function-parameter gadgets and hybrid gadgets at most, while memory errors with arbitrary read-write access can activate any x86 gadgets. To this end, we dynamically analyze the actual corruptibility of memory errors confirmed with concrete execution of PoC exploits. The data provide evidence of the prevalence of reusing existing CVEs to construct DOP attacks. For example, 5 out of the 6 stack-based buffer overflow vulnerabilities can use the assignment operations to achieve arbitrary write access (Column 3, 4 in Table VII). 8 out of 9 vulnerabilities enable arbitrary write capabilities with which the attacker can trigger a total of 1273 global gadgets (Column 13 in Table VII). In case of bitcoind, the attacker can only control local variables within the function using the CVE. Since we ignore the local gadgets in our analysis, we cannot simulate any MINDOP operations with this particular memory error.

Manually Confirmed Stitchability. Note that we have not checked each of 1273 gadgets against each CVE run to construct complete exploits — this is an onerous manual task. We have sampled a few cases and manually executed and verified if they are triggerable and stitchable using the CVE. The cases that we confirmed as exploitable by running the exploits and dynamically analyzing the execution are denoted by a check-mark (✓) in Table VII. We have also constructed end-to-end attacks using some of these gadgets as discussed in Section V-C.

B. Turing-Complete Examples

We have established that x86 data-oriented gadgets required to simulate MINDOP exist in real-world applications and can be triggered by the concrete vulnerabilities. Next we evaluate the ease of stitching multiple gadgets for building Turing-complete exploits. Currently we resorted to prioritizing cases and manually checking a random sample of gadgets based on their type and concrete memory errors. We present the details of two representative examples wherein the attacker: (1) actively interacts with the program, observes the behavior and crafts the next attack payload; (2) sends a single payload which triggers all the gadgets to execute the attacker's MINDOP program. Readers interested in end-to-end real attacks can read Section V-C first, where we show expressive attacks with these Turing-complete gadgets.

*1) **Interactive — ProFTPD**:* ProFTPD is a light-weight file server and its 1.2-1.3 versions have a stack-based buffer overflow vulnerability in the `sreplace` function (CVE-2006-5815 [45]). Line 14 in Code 7 shows the string copy

which overflows the stack buffer `buf`. We confirm that the dispatcher around this memory error and the data-oriented gadgets can be used to build Turing-complete calculation. We use the following methodology to implement MINDOP virtual operations with the x86 data-oriented gadgets in ProFTPD.

```
1  //memory error & assignment :
2  //      cmd_loop()->pr_cmd_dispatch()->_chdir()
3  //              ->pr_display_file()->sreplace()
4  char *sreplace(char * s, ...) {
5    char *src,*cp,**mptr,**rptr;
6    char *marr[33],*rarr[33];
7    char buf[PR_TUNABLE_PATH_MAX] = {'\0'};
8    src = s; cp = buf; mptr=marr; rptr=rarr;
9    ...
10   while (*src)
11     for (; *mptr; mptr++, rptr++) {          ...
12       //1st round: memory error
13       //2nd round: assignment
14       sstrncpy(cp,*rptr,blen-strlen(pbuf)); ...
15     }
16 }
```

Code 7. Code snippet of ProFTPD, with a stack-based buffer overflow. This code is also used to simulate the assignment gadget.

- *Conditional assignment operation.* We use the `sstrncpy` function to simulate an assignment which moves data from one arbitrary location to another arbitrary location. In the first iteration of the `while` loop (Line 10-15 in Code 7), the memory error corrupts the variable `cp` and content of the array `rarr`. So in the next iteration, both the source and the target of the string copy `sstrncpy` are controlled by the attacker. This way, the attacker simulates a MINDOP assignment operation. This gadget is conditional because the attacker can corrupt `src`, which is the condition for the second round of the loop body. If the condition is not satisfied, the assignment operation will not be executed.

```
1  //load : cmd_loop()->pr_cmd_dispatch()->_chdir()
2  //                          ->pr_display_file()
3  int pr_display_file(...) {...
4    outs = sreplace(p, buf, ...,
5       "%V", main_server->ServerName,);
6    pr_response_send_raw("%s-%s", code, outs);
7  }
8  void pr_response_send_raw(const char *fmt,...){
9    vsnprintf(resp_buf, size, fmt, msg);
10 }
```

Code 8. Simulated load gadget. This code copies data from a global variable `ServerName` to a global buffer `resp_buf`. With the assignment gadget that reads `*resp_buf` to `&ServerName`, we get the load gadget.

- *Dereference operations (Load / Store).* The load operation takes two memory addresses as input (say `p` and `q`) and performs operation `*p=**q`. We decompose the operation into two sub-operations: `*ptmp=*q` and `*p=*tmp`, such that the `ptmp` is the address of `tmp`. In ProFTPD, we use the assignment gadget to move data from the `resp_buf` to `&ServerName` as the first dereference. Then we use the function `pr_display_file` (Line 4, Code 8), which reads the content of `ServerName` to the buffer `resp_buf` as the second dereference. These two dereferences form a MINDOP load operation `*resp_buf=**resp_buf`. The MINDOP store operation is simulated by a similar method.

```
1  //addition : cmd_loop()->pr_cmd_dispatch()
2  //                        ->xfer_log_retr()
3  MODRET xfer_log_retr(cmd_rec *cmd) {
4    session.total_bytes_out += session.xfer.
         total_bytes;
5  }
```

Code 9. Simulated addition gadget. This code adds two fixed memory locations. Arbitrary memory addition can be achieved by combining this gadget with the assignment gadget.

- *Addition operation.* Code 9 shows the code in ProFTPD which adds two variables. The structure `session` is a global variable and hence all the operands of this gadget are the under attacker's control. To achieve an addition operation on arbitrary memory locations, we use the MINDOP assignment operation to load operands from desired source locations to the `session` structure, perform the addition, and then move the result to the desired destination location.

```
1  //dispatcher & jump :
2  void cmd_loop(server_rec *server,conn_t *c) {
3    while (TRUE) {
4      pr_netio_telnet_gets(buf, ..);
5      cmd = make_ftp_cmd(buf, ...);
6      pr_cmd_dispatch(cmd);  //calls functions
7             // with memory errors and gadgets
8    }
9  }
10 char *pr_netio_telnet_gets(char *buf,...) {
11   while(*pbuf->current != '\n' && toread>0)
12     //reads through virtual PC
13     *buf++ = *pbuf->current++;
14 }
```

Code 10. Gadget dispatcher and simulated jump gadget. `pbuf->current` is the virtual PC pointing to the malicious input.

- *(Conditional) jump operation.* Code 10 shows the ProFTPD program logic to read the next command from an input buffer. `pbuf->current` is a pointer to the next command in the input, thus forming a virtual PC for the attacker's MINDOP program. By corrupting `pbuf->current`, the attacker can select a particular input that invokes a specific MINDOP operation. We use the assignment operation to conditionally update the virtual PC, thus simulating a conditional jump operation.

To stitch these identified gadgets together, we identified a gadget dispatcher (Code 10) in the function `cmd_loop`. It contains an infinite loop that repeatedly reads requests from the remote attackers or cached in the buffer and dispatches the request to functions with various gadgets. For each request, the attacker embeds a malicious input which first exploits the memory error to prepare the memory state for one of these gadgets and then triggers the expected gadget to achieve the MINDOP operation. In Section V-C we show the case studies of expressive exploits against ProFTPD.

*2) **Non-interactive – Wireshark**:* Wireshark is a widely-used network packet analyzer and its versions before 1.8.0 have a stack-based buffer overflow vulnerability (CVE-2014-2299 [41]). The fixed-size

buffer pd (shown on Line 5 of Code 11) in function packet_list_dissect_and_cache_record accepts frame data from a mpeg trace file. If the attacker sends a malicious trace file containing a large frame (larger than 0xffff), the frame data overflows the buffer. This is used to overwrite variables col, cinfo, and parameter packet_list with malicious input. These corrupted values are then passed to the function packet_list_change_record which contains all the x86 data gadgets of our interest.

```
1  //vulnerable function
2  void packet_list_dissect_and_cache_record
3       (PacketList *packet_list, ...) {
4    gint col; column_info *cinfo;
5    guint8 pd[WTAP_MAX_PACKET_SIZE]; //vul buf
6    //memory error function
7    cf_read_frame_r(...,fdata,...,pd);
8    packet_list_change_record(packet_list,
9                          ...,col, cinfo);
10 }
11 //gadgets: assignment/load/store/addition
12 void packet_list_change_record(PacketList *
13     packet_list,..,gint col,column_info *cinfo)
14 {
15   record = packet_list->physical_rows[row];
16   record->col_text[col] =
17          (gchar *) cinfo->col_data[col];
18   if (!record->col_text_len[col])
19     ++packet_list->const_strings;
20 }
21 void gtk_tree_view_column_cell_set_cell_data(..)
22 {
23   for (cell_list = tree_column->cell_list;
24      cell_list; cell_list = cell_list->next) {
25     ....
26     //finally calls vulnerable function
27     show_cell_data_func();
28   }
29 }
```

Code 11. Wireshark code snippet of the vulnerable function and gadgets.

- *Assignment operation*. We identify an assignment operation from the function packet_list_change_record, called after the memory error function. Line 16 in Code 11 shows the gadget, where memory copy addresses are under the attack's control. col_text and col_data are of gchar ** type, so the assignment operation performs two dereferences per operand. To simulate a simple assignment from one memory location to another, the attacker corrupts record->col_text and cinfo->col_data. This is achieved by corrupting record and cinfo to point to controllable memory regions, where the value of record->col_text and cinfo->col_data will be retrieved.
- *Dereference operations (Load / Store)*. Line 16 in Code 11 also serves gadgets for simulating load and store operations of MINDOP, as it has two dereference operations. To simulate a load operation, the attacker corrupts record->col_text and cinfo. To simulate a store operation, the attacker can change the value of record and cinfo->col_data.
- *Conditional addition operation*. Lines 18-19 in Code 11 show a data gadget to perform a conditional increment operation. At each time this gadget is invoked it adds 1 to the target location. With the condition, we can implement an addition operation over arbitrary memory locations, where the attacker controls the condition as well as the operand of the increment.
- *Conditional jump operation*. The memory error is triggered by the file read, and the program maintains a file position indicator in the FILE structure. The attacker can change the file position indicator to force the program to non-linearly access the data frames in the file. This way the file position indicator serves as a virtual PC for the MINDOP program in Wireshark. Using the conditional addition operation, the attacker can simulate the MINDOP conditional jump operation by manipulating the file position indicator.

Since all the gadgets are executed after the memory error, each execution of the memory error can stably invoke at least one MINDOP operation. To chain a large number of gadgets together, we identify a gadget dispatcher from the parent function gtk_tree_view_column_cell_set_cell_data, as shown in Line 21-27, Code 11. In the first invocation of the memory error, the attacker uses the assignment operation to corrupt the loop condition cell_list, and points it to a fake linked-list in the malicious payload, making it an infinite loop. In each of the subsequent executions, the program reads malicious frame data to trigger different gadgets to synthesize the execution of expected MINDOP operations.

C. Why are Expressive Payloads Useful?

We demonstrate the stitchability of identified data-oriented gadgets by building concrete end-to-end exploits. We discuss three case studies to highlight the importance of expressive payloads. Specifically, we demonstrate how MINDOP empowers attackers to (a) bypass randomization defense without leaking addresses, (b) run a network bot which takes commands from attackers and (c) alter the memory permission.

*1) **Example 1 — Bypassing Randomization Defenses**:* Typical memory error exploits bypass Address Space Layout Randomization (ASLR) by mounting a memory disclosure attack to leak randomized addresses to the network [15]. But if the memory corruption vulnerability cannot leak / disclose the addresses then the attack fails. We show how to defeat ASLR with DOP without leaking any addresses to the network. As a real example, consider the vulnerable ProFTPD server, which internally uses OpenSSL for authentication. Our goal is to leak the server's OpenSSL private key. The program stores this key in a randomized memory region, so a direct access to it in presence of ASLR is not viable. We find that the private key has a chain of 8 pointers pointing the private key buffer, as shown in Figure 3. The locations of all the pointers except the base pointer are randomized; only the base pointer is reliably readable from the memory error. However, creating a reliable exploit needs to de-randomize 7 out of 8 pointers successfully to leak the key without any network disclosure of addresses!

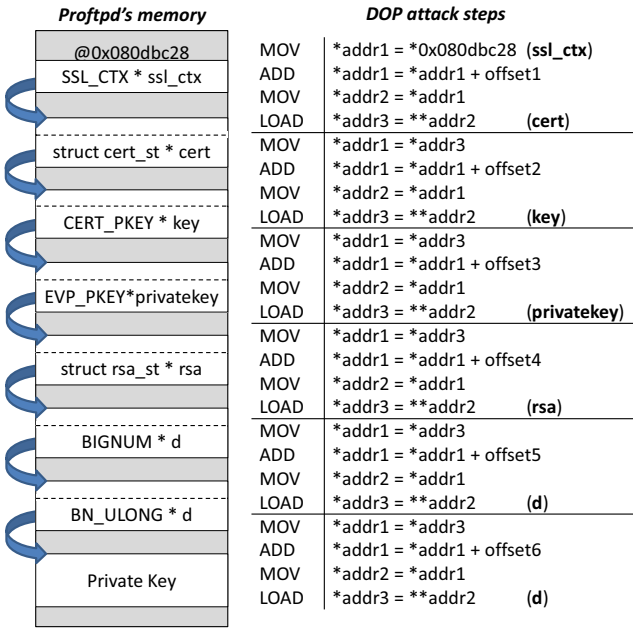

Fig. 3. Pointer dereference chain and malicious MINDOP program in attack against ProFTPD. The attack requires 8 memory dereferences from the deterministic location to the private key. Each dereference is implemented by 4 gadgets.

DOP is able to successfully construct such an attack. The key idea is to use a short MINDOP virtual program that starts from the base pointer (of known location) and dereferences it 7 times within the server's memory to correctly determine the randomized location of the private key. The virtual program needs to perform additions to compute the correct offsets within structures of intermediate pointers. In total, the virtual program takes 24 iterations, computing a total of 23 intermediate values to obtain the final address of the private key. Once we have the private key buffer's address, we simply replace an address used by a public output function, causing it to leak the private data to the network. We use the vulnerability CVE-2006-5815 [45] to simulate the malicious MINDOP program (as shown in Figure 3), and create an interactive DOP program that corrupts the program memory repeatedly. Each group of 4 gadgets performs one complete dereference operation. Note that `addr1`, `addr2` and `addr3` are fixed addresses in the gadgets. Therefore, the `MOV` between `ADD` and `LOAD` is necessary to deliver operands between operations.

Remark. TASR, a recent improvement in randomization defense, proposes to re-randomize the locations of code pointers frequently, such as on each network access system call (`read` or `write`) [14]. The primary goal of this defense is to reduce the susceptibility of commodity ASLR to address disclosure attacks. DOP can work even in the presence of such timely re-randomization because of two reasons. Firstly, TASR is applied to code pointers only, whereas our attack executes completely in the data-plane. Secondly, non-interactive DOP attacks can perform all the necessary computation in-place between two system-calls. For example, given simple programs

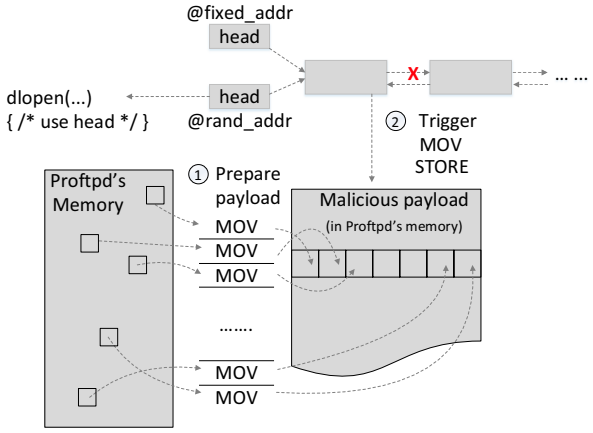

Fig. 4. Simulating a network bot. There are two steps in this attack: ① Prepare the payload in Proftpd's memory; ② Trigger the memory error. Each step uses many data-oriented gadgets.

in Code 12 and Code 13 (in Appendix B and Appendix C), TASR cannot defend it against DOP attacks. We refer interested readers to Appendix for details.

*2) **Example 2 — Simulating A Network Bot**:* One consequence of rich expressiveness in DOP exploits is that a vulnerable program can simulate a remotely-controlled network bot on the victim program. Though conceptually feasible, executing an end-to-end attack of such expressiveness requires careful design, which we illustrate in our concrete attack in ProFTPD.

ProFTPD invokes the `dlopen` function in its PAM module to dynamically load libraries. We analyzed `dlopen` to confirm that it has all the gadgets to simulate MINDOP (also see Shapiro *et al.* [48]). If the memory error allows the attacker to control a global metadata structure, ProFTPD provides the Turing-complete computation. In a normal execution, this metadata is loaded from a local object file; however, the remote attacker does not have the ability to create malicious object files on the server to misuse `dlopen`. To circumvent this, the attacker uses DOP to construct a first-stage payload in-memory and delivers it to `dlopen` which in turn executes the payload.

The main challenge in achieving this is ProFTPD's network input sanitization logic. It does not allow the attacker to directly supply the payload metadata object via a remote attack exploit. ProFTPD imposes several constraints on network inputs — inputs cannot contain a set of bytes such as zero, newline, and several other characters. To bypass these restrictions we build an interactive virtual program that serves as a first-stage payload. It sends malicious input that respects the program's constraints, and constructs the second-stage payload in the program's memory. It does so by copying the existing program memory bytes (instead of network input) to the payload address using MOV operations (Step 1 in Figure 4). In our end-to-end exploit, we perform over 700 interactions with the server to compose the malicious second-stage payload. Then we use movement and dereference operations to trigger the memory error (Step 2 in Figure 4). With these steps,

our exploit bypasses all the constraints on network inputs and enables arbitrary computation in the second-stage of the exploit. Finally, we force the program to invoke `dlopen` and execute the second-stage of the exploit. This simulates a bot that can repeatedly react to network commands sent to it remotely. We confirm that the bot performs arbitrary MINDOP program computation we request.

*3) **Example 3 — Altering Memory Permissions**:* Several control-flow and memory error defenses use memory page protection mechanisms as an underlying basis for defense. For instance, CFI defenses use read-only legitimate address tables to avoid metadata corruption [8] and DEP uses non-executable memory regions to prevent code injection attacks [16]. However, some critical functions, such as those in the dynamic linker, disable all memory protections temporarily to perform in-place address relocations. This gives the attacker a window of opportunity to violate the assumptions of the aforementioned defenses. To construct a successful exploit, the attacker utilizes the logic of the dynamic linker to corrupt the locations of its choice at runtime. The expressiveness of DOP is vital here — we have successfully built a second-stage exploit for `dlopen` (using a crafted metadata similar to Example 2 above) that permits arbitrary memory corruption or leakage of attacker-intended locations. We experimentally confirm that such exploits can bypass CFI implementations, like binCFI (utilizing read-only address translation tables [8]) or fixed-tag based solutions (assuming non-writable code region) [6], to effect control-hijacking exploits in ProFTPD. We refer interested readers to Appendix D for details.

D. Immunity against Control-Oriented Defenses

We have experimentally checked that our end-to-end exploits work when ASLR and DEP are enabled on the victim system. All 3 attacks work without reporting any error or warning. The first attack successfully sends the server's private key to the malicious client. For the second attack, we confirm that the bot performs arbitrary MINDOP program computation we request. While for the third attack, we modify the code section (provided by DEP) to start a shell in the server process.

VI. DISCUSSION

We have shown that DOP exploits can create semantically expressive payloads without violating the integrity in the control plane. In this section we discuss their implication, in particular, the effectiveness in re-enabling control-hijacking exploits and possible defenses to mitigate them.

A. Re-Enabling Control-Hijacking Attacks

A natural question is whether DOP can undermine control-flow defenses to re-enable attackers to perform control-hijacking attacks. First, our results have shown that bypassing commodity ASLR is feasible, without the need for memory disclosures. Commodity ASLR implementations randomize memory segments at the start of the application [15]. Newer defenses propose to re-randomize the program memory periodically, say at certain I/O system calls, thereby increasing resistance to disclosed addresses. One such proposal, called TASR [14], restricts randomization to code pointers. As we have discussed, it can be bypassed using a non-interactive DOP attack. Code randomization defenses typically aim to prevent ROP gadgets from either preventing their occurrence or randomizing their locations. If these defenses rely on keeping secret metadata in memory, then DOP offers a way to bypass protection. However, some randomization techniques do not make such assumptions [49], and conceptually not bypassable by DOP attacks.

A number of solutions for enforcing control-flow integrity have been proposed. Some of them rely on memory page permission to protect code integrity or metadata integrity. For example, Abadi *et al.* uses non-writable target IDs in memory to identify legitimate control transfer targets [6]. BinCFI relies on non-writable address translation table to enforce target checking [8]. DOP attacks can corrupt such non-writable IDs, or non-writable translation tables, and thus re-enable code reuse attacks. More seriously, DOP can directly modify non-writable code region to re-enable code-injection attacks, as we discuss in Section V-C3. Even if the IDs values are randomized to avoid effective guessing (an alternative to read-only IDs), DOP can still read the ID content and reuse it to build "legitimate" code blocks.

Furthermore, a class of defenses aims to protect integrity of code pointers, either via cryptographic techniques or via memory isolation and segmenting [12], [13]. Code pointer integrity (CPI) is one such defense, which is based on memory isolation [13]. CPI is designed to isolate code pointers and data pointers that eventually point to code into another protected memory region. Since the defense accounts for data pointers, one way to bypass it is to break the isolation primitives (e.g., SFI [50]). Conceptually, DOP attacks so far have not yet been able to demonstrate such capability. Cryptographicaly enforced CFI (CCFI) is another technique which cryptographically protects code pointers [12]. The authors acknowledge that protecting data pointers that may point to code pointers is important for achieving control-flow safety; however, this is left out of scope of the paper's proposals. DOP attacks can easily change data pointers that point to code pointers to violate CFI if they are left unprotected. We have checked the possibility for a simple proof-of-concept program against the CCFI implementation (details in Appendix B).

Finally, we point out that our discussion here pertains to explicitly subverting the goals of control-flow hijacking defenses. If subverting control-flow is not the goal, DOP executes in the presence of all such defenses without disturbing any control-flow properties or code pointers. We have experimentally checked for these in Section V-D.

B. Potential Defenses for DOP

*1) **Memory Safety**:* Memory safety prevents memory errors in the first place, by detecting any malicious memory corruption. For example, Cyclone and CCured introduce a safe type system to the type-unsafe C language [51], [52]. SoftBound with CETS stores metadata for each pointer inside a disjointed

memory for bound checking and identifier matching to force a complete memory safety [53], [54]. Cling enforces the temporal memory safety through type-safe memory reuse [55]. Data-oriented programming utilizes a large number of memory errors to stitch various data-oriented gadgets. Hence a *complete* memory safety enforcement will prevent all possible exploits, including DOP. However, high performance overhead prevents the practical deployment of current memory safety proposals. For example, SoftBound+CETS suffers a 116% average overhead. Development of practical memory safety defense is an active research area [56].

*2) **Data-Flow Integrity**:* Data-flow integrity (DFI) generates the static data-flow graph (DFG) before the program execution [57]. The data-flow graph is a database of define-use relationship. DFI instruments the program to check whether each memory location is defined by legitimate instructions before read operations. This way DFI prevents malicious define behaviors that corrupt program memory. Recent work uses DFI to protect kernel security-critical data [58]. A complete enforcement of data flow integrity in all memory regions can mitigate data-oriented programming. However complete DFI has a high performance overhead (44% for intraproc DFI and 103% for interproc DFI). Note that selective protection on security-critical data [58] may work on DOP, as it protects some pieces of data, but not a panacea for all data.

*3) **Fine-grained Data-Plane Randomization**:* We have shown in Section V-C that coarse-grained randomization or randomization on code region cannot prevent DOP attacks. Fine-grained data-plane randomization can mitigate DOP attacks as DOP still needs to get the address of some non-control data pointers [59], [60]. For example, to stitch one gadget with another, DOP corrupts the store address of the first gadget or the load address of the second gadget to make them the same. However, a fine-grained randomization on data-plane may occur a high performance overhead as all the data (both control- and non-control- data) should be randomized frequently. A data-plane randomization with high performance and strong security guarantee is still an open question.

*4) **Hardware and Software Fault Isolation**:* Memory isolation is widely used to prevent unauthorized access to high-privileged resources. Only legitimate code region has access to particular variables. This can be used to prevent unexpected access to security-critical data, like user identity. This way it can prevent some direct-data-corruption attacks [19], [22]. However DOP does not rely on the availability of security-critical data – it can corrupt pointers only to stitch data-oriented gadgets. To prevent such attacks, memory isolation has to protect all pointers from pure data. However, it is challenging to accurately identify pointers. Further there are numerous pointers in the program. Protecting all of them will introduce high-performance overhead. Therefore isolation only prevents a part of data-oriented programming attacks when the program is correctly protected with pointer isolation.

VII. Related Work

In Section VI, we have discussed potential defenses against control-hijacking attacks and non-control data attacks. In this section, we focus on the most closely related work.

Non-Control Data Attacks. In Section II-A, we have an in-depth discussion on non-control data attacks, including Chen *et al.* [19], control-flow bending attack [25] and FLOWSTITCH [22]. The difference between DOP and previous work is that DOP does not rely on any specific security-critical data or functions, like system call parameters or printf-like functions. Instead, it only reuses abundant data-oriented gadgets to build expressive attacks. Due to this feature, it is more challenging to prevent DOP attacks. Simple defenses mechanisms can sanitize critical data at particular program locations with acceptable performance overhead. But protecting all data pointers will introduce extremely high overhead.

Return-Oriented Programming. ROP technique and its variants have been extensively explored recently [1]–[5], [23], [61], [62]. For example, counterfeit object oriented programming (COOP) demonstrates that Turing-complete attacks can be built with only virtual function calls in C++ [62]. However, ROP attacks change the control flow of the vulnerable program, which can be mitigated by rapidly advancing control-flow integrity solutions [6]–[8], [10], [11], [17], [18]. In contrast, data-oriented programming manipulates variables in the data plane and keeps the original control-flow. It works even when advanced control-flow defenses are deployed.

Turing-Complete Weird Machines. Several work exploits auxiliary features in software to provide Turing-complete computation, called *weird machines*. Shapiro *et al.* used the dynamic loader on Linux system to provide such computation ability [48], which is used by DOP to build further attacks. Other weird machines can be built with DWARF (Debugging With Attribute Records Format) bytecode [63], the page fault handling mechanism [64] or the DMA (direct memory access) component [65]. DOP demonstrates the Turing-completeness in the data plane of arbitrary x86 programs.

VIII. Conclusion

In this paper, we show that with a single memory error, non-control data attack can mount Turing-complete computations using data-oriented programming. Our experiments on 9 real-world applications show that data-oriented gadgets and gadgets dispatchers required for DOP are prevalent. We build 3 end-to-end attacks to demonstrate the practical implications of not protecting the program's data-plane.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. Thanks to Yaoqi Jia for his help on our experiments. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. This work is supported in part by a research grant from Symantec.

REFERENCES

[1] 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*, 2007.

[2] 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*, 2011.

[3] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-Oriented Programming Without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[4] E. Bosman and H. Bos, "Framing Signals - A Return to Portable Shellcode," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[7] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[8] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[9] 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 *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[10] B. Niu and G. Tan, "Per-Input Control-Flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[11] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[12] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: Cryptographically Enforced Control Flow Integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[13] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer Integrity," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[14] 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*, 2015.

[15] PaX Team, "PaX Address Space Layout Randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt, 2003.

[16] Microsoft, "Data Execution Prevention (DEP)," 2006, http://support.microsoft.com/kb/875352/EN-US/.

[17] B. Niu and G. Tan, "Modular Control-flow Integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[18] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[19] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[20] "The Heartbleed Bug," http://heartbleed.com/.

[21] Yu Yang, "ROPs are for the 99%, CanSecWest 2014," https://cansecwest.com/slides/2014/ROPs_are_for_the_99_CanSecWest_2014.pdf, 2014.

[22] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic Generation of Data-Oriented Exploits," in *Proceedings of the 24th USENIX Security Symposium*, 2015.

[23] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size Does Matter in Turing-complete Return-oriented Programming," in *Proceedings of the 6th USENIX Conference on Offensive Technologies*, 2012.

[24] "SOP Bypass Demos," https://goo.gl/4PfXEg.

[25] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the 24th USENIX Security Symposium*, 2015.

[26] C. Planet, "A Eulogy for Format Strings," http://phrack.org/issues/67/9.html.

[27] Microsoft, "_set_printf_count_output," https://msdn.microsoft.com/en-us/library/ms175782.aspx.

[28] "ProFTPD — Highly configurable GPL-licensed FTP server software," http://www.proftpd.org/.

[29] "WU-FTPD Server," http://www.wu-ftpd.org/.

[30] "Nginx HTTP Server 1.3.9-1.4.0 Chunked Encoding Stack Buffer Overflow," http://mailman.nginx.org/pipermail/nginx-announce/2013/000112.html.

[31] "Bitcoin - Open source P2P money," https://bitcoin.org/.

[32] "SSH Communications Security," www.ssh.com/.

[33] "Wireshark Go Deep." https://www.wireshark.org/.

[34] "Glibc - Gnu," https://www.gnu.org/s/libc/.

[35] "BusyBox," www.busybox.net.

[36] "musl libc," http://www.musl-libc.org/.

[37] "MCrypt," mcrypt.sourceforge.net/.

[38] "Sudo Main Page," http://www.sudo.ws/.

[39] "Buffer Overflow Vulnerability in UPnP library used by Bitcoin Core," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-6031.

[40] "musl libc inet_pton.c Remote Stack Buffer Overflow Vulnerability," https://security-tracker.debian.org/tracker/CVE-2015-1817.

[41] "Stack-Based Buffer Overlfow in the MPEG parser in Wireshark," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2299.

[42] " nginx: Stack-based Buffer Overflow," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028.

[43] "Stack-Based Buffer Overflow in Mcrypt," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4409.

[44] "Sudo Format String Vulnerability," http://www.sudo.ws/sudo/alerts/sudo_debug.html, 2012.

[45] "Stack-Based Buffer Overlfow in the sreplace Function in Proftpd," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5815.

[46] "Integer Overflow Vulnerability in SSH CRC-32 Compensation Attack Detector," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144.

[47] "Wu-Ftpd Remote Format String Stack Overwrite Vulnerability," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0573.

[48] R. Shapiro, S. Bratus, and S. W. Smith, ""Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata," in *Proceedings of the 7th USENIX Conference on Offensive Technologies*, 2013.

[49] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[50] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-based Fault Isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993.

[51] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," in *Proceedings of the USENIX Annual Technical Conference*, 2002.

[52] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe Retrofitting of Legacy Code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[53] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," in *Proceedings of the 30th ACM SIG-PLAN Conference on Programming Language Design and Implementation*, 2009.

[54] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: Compiler Enforced Temporal Safety for C," in *Proceedings of the 9th International Symposium on Memory Management*, 2010.

[55] P. Akritidis, "Cling: A Memory Allocator to Mitigate Dangling Pointers," in *Proceedings of the 19th USENIX Security Symposium*, 2010.

[56] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[57] M. Castro, M. Costa, and T. Harris, "Securing Software by Enforcing Data-Flow Integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[58] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee, "Enforcing Kernel Security Invariants with Data Flow Integrity," in *Proceedings of the 23th Annual Network and Distributed System Security Symposium*, 2016.
[59] S. Bhatkar and R. Sekar, "Data Space Randomization," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.
[60] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A Practical Approach for Adaptive Data Structure Layout Randomization," in *Proceedings of the 20th European Symposium on Research in Computer Security*, 2015.
[61] 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 *Proceedings of the 34th IEEE Symposium on Security and Privacy*, ser. SP '13, 2013.
[62] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.
[63] J. Oakley and S. Bratus, "Exploiting the Hard-working DWARF: Trojan and Exploit Techniques with No Native Executable Code," in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, 2011.
[64] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, "The Page-fault Weird Machine: Lessons in Instruction-less Computation," in *Proceedings of the 7th USENIX Conference on Offensive Technologies*, 2013.
[65] M. Rushanan and S. Checkoway, "Run-DMA," in *Proceedings of the 9th USENIX Conference on Offensive Technologies*, 2015.

APPENDIX

A. Example of a Transition Table in MINDOP

Table VIII shows one example of transition table for simulating a Turing machine that shifts the given input by one bit.

TABLE VIII
Transition table for a Turing machine that shifts the binary input by one bit (equivalent to SHL instruction).

Q \ Σ	$S_{cur}=0$			$S_{cur}=1$			$S_{cur}=\sigma_0$		
	q_{nxt}	S_{nxt}	D	q_{nxt}	S_{nxt}	D	q_{nxt}	S_{nxt}	D
q_0	q_1	0	1	q_1	1	1	q_0	σ_0	1
q_1	q_1	0	1	q_1	1	1	q_2	0	0
q_2	q_3	0	1	q_3	1	1	q_3	σ_0	1
q_3	HALT	-	-	HALT	-	-	HALT	-	-

B. A Program Allowing DOP to Bypass TASR and CCFI

```
1  typedef struct _mystruct {
2      void (*foo)();
3  } mystruct;
4
5  void m1() = { printf(``hello from m1''); }
6  void m2() = { printf(``hello from m2''); }
7
8  mystruct ms1 = { .foo = m1 };
9  mystruct ms2 = { .foo = m2 };
10 mystruct *pms1 = &ms1, *pms2 = &ms2;
11
12 int main(int argc, char * argv[]) {
13     int old_value, new_value;
14     int *p = &old_value, *q = &new_value;
15     char buf[64];
16
17     memcpy(buf, argv[1]);  // memory error
18     *p = *q;               // assignment gadgets
19
20     pms1.foo();
21 }
```

Code 12. A simple program that enables DOP to build control attacks even if TASR and CCFI are in place, as the data pointers are not protected.

Code 12 shows a simple program where TASR and CCFI cannot prevent the control attack built with the help of DOP, as the data pointers are not protected by TASR or CCFI [12], [14]. There are two data pointers, `pms1` and `pms2` and they are also pointing to code pointers `foo` in corresponding structures. Legitimately, the indirect function call in line 20 will call the function `m1`. With the memory error in line 17 and the assignment gadget in line 18, attackers can construct non-control data exploit to swap the value of `pms1` and `pms2`. Then the code in line 20 will call function `m2` instead.

C. Another Program Allowing DOP to Bypass TASR

```
1  Disassembly of section .plt
2
3  0804ada0 <system@plt>:
4                 jmp *0x08104000
5  0804adb0 <setuid@plt>:
6                 jmp *0x08104004
7  0804adc0 <read@plt>:
8                 jmp *0x08104008
9
10 int server(int sockfd) {
11     int old_value, new_value;
12     int *p = &old_value, *q = &new_value;
13     int connect_limit = 100;
14     char buf[64];
15
16     while(connect_limit--) {
17         read(sockfd, buf);  // memory error
18         *p = *q;            // assignment gadgets
19     }
20 }
```

Code 13. A simple program that enables DOP to break TASR, as no `write` operation is necessary during attack.

Code 13 shows a piece code to illustrate another method to bypass TASR with DOP. This code invokes library functions, like `system` and `setuid`, thus having the function addresses in the `.plt` section. With the assignment gadget in line 18, the attackers can copy the function addresses (e.g., addresses of `setuid` and `system`) from the `.plt` section to the selected memory region, like the stack location for `server` return address. The gadget dispatcher in line 16 and 17 enables attackers to prepare a complete stack context for a return-to-libc attack. When the function returns, the return-to-libc attack will be launched. TASR fails to prevent such attack as attackers use DOP to prepare the payload on the stack in the memory, without any address leakage through the `write` system call.

D. Using DOP to Break CFI Implementations and DEP

BinCFI. BinCFI uses a read-only table to store all legitimate function entries and call-sites [8]. Each function call is allowed to jump to any function entry, and each function return is permitted to return to any call-site. A successful BinCFI attack should lead the program call / return to arbitrary locations. We show one vulnerable program in Code 14 that allows DOP to mount a BinCFI attack. With the memory error in line 15, attackers can deliver malicious relocation metadata on the stack and change the value of `p` and `q`. With the store gadget

```
1  typedef struct _mystruct {
2      void (*foo)();
3  } mystruct;
4
5  void m1() = { printf(``hello from m1''); }
6
7  mystruct ms1 = { .foo = m1 };
8  mystruct *pms1 = &ms1;
9
10 int main(int argc, char * argv[]) {
11     int old_value, new_value;
12     int *p = &old_value, *q = &p;
13     char buf[64];
14
15     memcpy(buf, argv[1]);  // memory error
16     **q = *p;              // store gadgets
17     *p = *q;          // assignment gadgets
18
19     dlopen("mylib.so");
20     pms1.foo();
21 }
```

Code 14. A simple program that enables DOP to build control attacks to bypass Bin-CFI and non-writable-tag based CFI.

in line 16, attackers can change the `link_map` structure link list to make it link the malicious relocation metadata. One functionality of `dlopen` is to patch the relocated addresses before real execution. This makes `dlopen` able to modify arbitrary memory location, even code pages or read-only data sections. When `dlopen` is invoked, the malicious metadata will trigger the `dlopen`'s internal gadgets to corrupt the read-only table. By adding expected code addresses into the table, attackers is allowed to make the execution jump to arbitrary code region (line 19).

Protections based on Non-Writable Code Section. W⊕X disallows the write permission on code section, or execute permission on data section, to protect code integrity [16] and control-flow integrity. For example, the CFI proposed by Abadi *et al.* relies on read-only tags inside code region to enforce the security check [6]. Specifically, it places tags before legitimate control-flow transfer targets and checks the target tag with the predefined tag before each control-flow transfer at runtime. DOP can help break defenses that are based on non-writable code. First, attackers can use DOP to invoke `dlopen` to corrupt arbitrary code region to mount code injection attacks. Note that the data region is still non-executable, even with DOP attacks. Second, DOP can help change the CFI tags in code region to bypass the CFI solution. Attackers either copy the tag from the legitimate code target to the illegal location, or just overwrite both the checking code to disable CFI.