CORE
Metadata, citation and similar papers at core.ac.uk
Provided by ScholarBank@NUS

SYSTEMATIC METHODS FOR MEMORY ERROR DETECTION AND EXPLOITATION

HU HONG

(*B.Eng., Huazhong University of Science and Technology*)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2016

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

HU HONG

13 January 2016

ACKNOWLEDGEMENTS

I am grateful to have my PhD study in National University of Singapore. During this long journey, many people provided their generous supports and suggestions. Without them, I will not harvest so much, including this thesis.

First, I thank my advisor, Prof. Zhenkai Liang, who leads me to the exciting topic of binary analysis and guides me to be a security researcher with his insight and experience. Along the challenging journey, he always advised me to think questions at a high level and dive to deep technical details, with great patience and perseverance. Thanks to Prof. Prateek Saxena for insightful comments on several papers that constitute this thesis. I learned a lot from the discussion and collaboration with him. Thanks to several professors who gave me valuable feedback to my thesis: Prof. Abhik Roychoudhury, Prof. Roland Yap and Prof. Heng Yin. They have expanded my research view and help me find the shortcomings of my work.

I am indebted to all of my collaborators for their hard working and strong support. The ideas inside this thesis were inspired from discussions with them. The solid result of each project witnesses our creative collaboration. I enjoyed working with Adrian Sendroiu, Zheng Leong Chua, Shweta Shinde, Xinshu Dong, Xiaolei Li, Li Li, Guangdong Bai and Yaoqi Jia.

Many thanks to all my lab-mates, from whom I have learned broadly in research and daily life: Kailas Patil, Ting Dai, Mingwei Zhang, Chunwang Zhang, Sai Sathyanarayan, Bodhisatta Barman Roy, Enrico Budianto, Shruti Tople, Ziqi Yang, Loi Luu, Hung Dang and many others.

Thanks to my family for their selfless support. My mother and farther have worked so hard to support my long journey. My sister carefully takes care of my parents, providing me the freedom to pursuit my dreams. Finally, special thanks to my beloved fiancée, Dongyan Zhang, who has persistently encouraged and accompanied me for the hard times.

Contents

Summary vii

List of Tables ix

List of Figures xi

1 Introduction 1

1.1 Thesis Overview . . . 4

2 Background 7

2.1 Memory Error & Detection . . . 7

2.1.1 Memory Error Detection . . . 8

2.1.2 Memory Safety Enforcement . . . 9

2.2 Exploits and Defenses . . . 10

2.2.1 Memory Error Exploits . . . 11

2.2.2 Defenses against Exploits . . . 13

2.3 The Need for New Systematic Solutions . . . 18

3 Detecting Memory Errors in Privilege-Separated Software 19

3.1 Problem . . . 22

3.1.1 Motivating Example . . . 22

3.1.2 Problem Definition . . . 23

3.1.3 Memory Access Patterns to Detect DUIs . . . 24

3.2 Design . . . 25

3.2.1 Overview . . . 25

3.2.2 Suspicious Instruction Shortlisting . . . 26

3.2.3 Dereference Behavior Analysis . . . 27

3.3 Implementation . . . 30

3.3.1 Taint Propagation 30
3.3.2 Access Formula Generation 31
3.4 Evaluation 32
3.4.1 Efficacy 32
3.4.2 Performance 37
3.4.3 Security Implications 37
3.5 Discussion 38
3.6 Related Work 39
3.7 Summary 40

4 Exploiting Memory Errors with Data-Flow Stitching **41**

4.1 Problem Definition 44
4.1.1 Motivating Example 44
4.1.2 Objectives & Threat Model 46
4.1.3 Problem Definition 47
4.1.4 Key Technique & Challenges 48
4.2 Data-Flow Stitching 50
4.2.1 Basic Stitching Technique 51
4.2.2 Advanced Stitching Technique 52
4.2.3 Challenges from ASLR 56
4.3 The FLOWSTITCH System 60
4.3.1 Memory Error Influence Analysis 61
4.3.2 Security-Sensitive Data Identification 62
4.3.3 Stitching Candidate Selection 63
4.3.4 Candidate Filtering 64
4.4 Implementation 65
4.5 Evaluation 66
4.5.1 Efficacy in Exploit Generation 66
4.5.2 Reduction in Search Space 68
4.5.3 Performance 69
4.5.4 Case Studies 70
4.6 Related Work 72

4.7 Summary 73

5 Exploiting Memory Errors with Data-Oriented Programming **75**

5.1 Problem 78

5.1.1 Background: Data-Oriented Attacks 78

5.1.2 Example of Data-oriented Programming 79

5.1.3 Research Questions 81

5.2 Data-Oriented Programming 81

5.2.1 DOP Overview 81

5.2.2 Data-Oriented Gadgets 82

5.2.3 Gadget Dispatcher 85

5.2.4 MINDOP is Turing-Complete 88

5.3 DOP Attack Construction 91

5.3.1 Challenges 91

5.3.2 Gadget Identification 92

5.3.3 Dispatcher Identification 94

5.3.4 Attack Construction 95

5.4 Evaluation 96

5.4.1 Feasibility of DOP 96

5.4.2 Turing-Complete Examples 101

5.4.3 Why are Expressive Payloads Useful? 106

5.4.4 Immunity against Control-Oriented Defenses 110

5.5 Discussion 110

5.5.1 Re-Enabling Control-Hijacking Attacks 110

5.5.2 Example Vulnerable Programs 111

5.5.3 Potential Defenses for DOP 114

5.6 Related Work 116

5.7 Summary 117

6 Conclusion **119**

Bibliography **121**

Summary

Memory errors are persistent threats to computer systems. Attacks exploiting memory errors have resulted in severe damage in real-world programs. Every year more than one thousand of memory errors are detected and reported. At the same time, exploit mechanisms are rapidly evolving, enabling attacks to bypass known protections. To prevent the huge damage from memory errors, it is crucial for us to detect them in advance and predict their evolving trend.

In this thesis, we use systematic methods to detect memory errors and discover new attack mechanisms. We propose and implement three novel methods. In the first work, we detect memory errors shown in the privilege-based isolation. In particular, we identify the arbitrary memory access vulnerability, where the unexpected memory access can cross different isolated partitions. We build a systematic way to detect the arbitrary memory access code and report its severity. Then we look into new exploit mechanisms that can bypass most of the known defense mechanisms – the data-oriented attack. Specifically, we study the consequence of data flow re-construction, where original data flows in a program are split into small pieces and are reconnected by memory errors. We propose a novel method, called *data-flow stitching*, to connect disjoint data flows to either leak information or gain extra privileges. Data-flow stitching significantly enlarges the capability of data-oriented attacks. Finally, we explore the expressiveness of data-oriented attacks regarding the ability to perform arbitrary computations. We propose a new exploit construction technique, called *data-oriented programming (DOP)*, to selectively stitch basic data-flow gadgets for a desired purpose. With DOP, we build Turing-complete data-oriented attacks resulted from common memory errors, demonstrating the strong expressiveness.

List of Tables

3.1 Performance of the DUI Detector 37

4.1 Deterministic memory region size 57

4.2 Experiment environments and benchmarks. 66

4.3 Evaluation of FLOWSTITCH 67

4.4 FLOWSTITCH performance 69

5.1 Simulating a loop with DOP attack 81

5.2 MINDOP language 82

5.3 Example data-oriented gadget of addition operation 84

5.4 Example data-oriented gadget of assignment operation. 85

5.5 Example data-oriented gadget of dereference operation 85

5.6 Transition table for a Turing machine that shifts the binary input by one bit (equivalent to SHL instruction). 90

5.7 Prevalence of DOP gadgets & dispatchers 97

5.8 Reachability and corruptibility of x86 gadgets in the presence of a specific memory error 99

List of Figures

3.1 Design of the DUI Detector 25

4.1 An example 2D-DFG of Code 4.1 48

4.2 A data-oriented attack of Code 4.1 49

4.3 Single-edge stitch of wu-ftpd 50

4.4 Two-edge stitch of wu-ftpd 53

4.5 Stitch edge selection 55

4.6 Stitch with deterministic memory addresses of orzhttpd 58

4.7 Stitch by complete memory address reuse of sudo 60

4.8 Overview of FLOWSTITCH 60

5.1 Malicious input to mount DOP attack 80

5.2 Design model of DOP in MINDOP 86

5.3 Pointer dereference chain and malicious MINDOP program in attack against ProFTPD 107

5.4 Simulating a network bot with DOP attacks 108

Chapter 1

Introduction

Memory errors are classical problems in computer system security. At a specific execution time, one particular memory operation should only access a subset of the full memory space by design. However, memory errors lead to *unexpected* memory accesses regarding the location or the time. For example, a process with buffer overflow [61,132] can visit the memory beyond the boundary of the expected buffer; a process with use-after-free [16] can access a freed variable. Under memory errors, attackers can cause predictable and controllable results with elaborated inputs, like control of a remote system (e.g., through Ghost [69]), or leakage of the sensitive information (e.g., through Heartbleed [13]). Many exploit methods have been developed to build reliable attacks, such as code injection attack [132], return-to-libc attack [125] and return-oriented programming (ROP) [149].

Low-level programming languages, like C/C++, provide a large breeding ground for memory errors. These languages contain a set of programming APIs for developers to access the low-level memory, like memory allocation (e.g., `malloc` in C) and release (e.g., `free` in C). As programmers can optimize the memory usage for better performance, these languages get wide adoption during the real-world program development. Many popular systems, including browsers (e.g., Chrome and Internet Explorer) and operating systems (e.g., Linux and Windows), are developed in C/C++. However, the freedom of the low-level memory access is a double-edged sword: convenient APIs help developers program efficiently, but any programming mistakes can lead to severe memory errors. For example, if the program inadvertently dereferences memory with a freed object pointer, it leads to a use-after-free memory error. As the code base is

getting larger, it is more and more difficult for developers to prevent memory errors.

To mitigate the damage by memory errors, a wide range of detection and defense mechanisms have been proposed. Memory safety is designed to provide safe memory access. It tries to prevent memory errors in the first place. Various implementations of memory safety [17, 18, 105, 109, 113, 122–124, 141] bring type systems to unsafe languages or check the memory access bound and the life-cycle at run-time. However, these implementations often suffer from a high performance overhead [105, 122–124], or only provide partial protection of control data [109, 113], rendering the current solutions impractical. As a second-line defense, exploit prevention mechanisms are developed to prevent a successful exploit, even if attackers can corrupt the memory with errors. Memory isolation [34,42,106,138,168] aims to protect high-privileged resources by separating unprivileged resources and high-privileged resources into different partitions. Data execution prevention (DEP) [21] removes memory regions that are both executable and writable, so that attackers cannot inject any new code into the vulnerable program. It effectively prevents pure code injection attacks. Address space layout randomization (ASLR) [20, 26, 30, 32, 92, 100, 167] makes the program address highly unpredictable. Attackers have no knowledge about the target to divert the control flow, and thus likely fail to build a successful attack. Control flow integrity (CFI) [15] draws on the program's benign control flow graph (CFG) to define all legitimate control flow transfer targets and requires the program execution to conform to the CFG. It limits every indirect control flow transfer to one of the legitimate targets. Data flow integrity (DFI) [50, 154] only allows memory updates by instructions in the legitimate instruction set. More efforts [58, 63, 82, 129, 133, 147, 158, 160, 170, 173, 178, 180, 182] have been spent to improve the security and practical viability of memory error solutions. These methods significantly raise the bar of successful memory error exploits.

However, exploit methods evolve rapidly to bypass public-known defenses. Even known memory errors can cause significant threats with new exploits. For example, code reuse attacks, like return-to-libc and ROP, can circumvent the DEP defense. Information leakage [13, 148] and heap spray [156, 161] are leveraged to bypass defense mechanisms based on the address randomization. The arms race between exploits and defenses will not end in the near future as attackers always seek new ways for successful exploits. Defenders need to deploy a proactive defense to prevent new attacks. This

requires researchers to understand the trend of the exploit development and predict the next possible methods. Currently one of the most advanced defenses is control flow integrity (CFI), which tries to prevent control flow hijacking attacks. Researchers have spent a lot of efforts to make CFI efficient and effective so that they can be deployed in real world systems [121, 160]. But the memory error is still dangerous if new exploits do not modify any control data (like non-control data attack [56]). Currently our knowledge of attacks on non-control data is limited. However several recent attacks [13, 175] have demonstrated their severe damage. It is urgent for us to systematically understand the capability of non-control data attacks so that we can deploy defense mechanisms in advance. In this thesis, we aim to answer the following research questions: How to systematically detect unexpected memory accesses, which are the sources of memory errors? What are possible methods to build non-control data attacks (or data-oriented attacks), beyond the simple non-control data corruption? What is the computational expressiveness of data-oriented attacks? Can they be Turing-complete, as control-flow hijacking attacks (like return-to-libc and ROP)?

Thesis Statement. *We systematically detect memory errors where attackers control the memory access. We propose and build novel methods to discover and analyze expressive data-oriented attacks.*

Our Approach. We propose systematic analysis to understand the program's memory access behavior, and identify memory errors and exploits. We first detect memory errors exposed during the privilege-based isolation. Privilege-based isolation is designed to provide more secure programs, where program components are separated into different partitions. However, isolation itself is not adequate to prevent the malicious cross-partition memory access. We develop a tool to systematically detect vulnerabilities that allow attackers to bypass isolation methods by controlling inputs to isolated partitions. Then we look into the new exploit vector in the data space, which consists of memory variables not directly used in control-flow transfer instructions. The state-of-the-art attacks in data space directly corrupt security-critical data. Instead, we take data flows of a program, which describe how the data are used, as the attack targets. We observe that end-to-end data flows can be broken into small pieces with the memory error. We

propose a general technique to build data-oriented attacks, called data-flow stitching, by connecting particular data flows of security-critical data. We show that data-flow stitching is practical and significantly enlarges the space of data-oriented attacks. Finally, we explore the computational expressiveness of data-oriented attacks, regarding the ability of performing arbitrary computations. The conventional wisdom is that data-oriented attacks can only corrupt or leak several bytes of the critical data, due to its limitation on control flow diversion. We perform a comprehensive evaluation on the prevalence of the fundamental elements to build high expressive data-oriented attacks. Our result shows that data-oriented attacks can express Turing-complete calculations, even starting from one common memory error, like buffer overflow.

1.1 Thesis Overview

Next, we give an overview of the projects constituting this thesis.

A Systematic Method to Detect Memory Errors Exposed by Privilege-based Isolation. First we study the consequence of the execution environment change where memory access components are disconnected from original constraints. This disconnection is introduced by privilege-based program isolation, a fundamental method to improve the security of the legacy code [34, 42, 106, 168]: the monolithic code is divided into several partitions. Each partition is protected through memory isolation, where others cannot directly access one partition's memory space, unless with clearly defined interfaces. However, component isolation re-positions each isolated partition into a "relaxed" execution environment with less program constraints (e.g., input sanitization). The "relaxed" memory access operations obtain extra capability. Particular memory access patterns inside one partition may allow untrusted parties to arbitrarily influence its private memory space through the interfaces. For example, the Iago [54] attack allows the malicious kernel to corrupt program's private memory space, even if the program is protected by isolation [57]. We refer to this type of memory access patterns as *Dereference Under the Influence (DUI)*. The DUI exploit provides attackers a feasible channel to launch attacks against memory isolation. In this thesis, we develop a systematic method to analyze each individual memory access component in the monolithic program to detect potential memory errors. Our method successfully detected

several DUI vulnerabilities in the privilege-separated software. Further, it reports the attackers' capability obtained by DUI exploits.

Memory errors provide attackers the ability to mount exploits. Many research efforts have been spent on the prevention of control flow hijacking attacks. However, attackers are not limited to one particular exploit methods. Data-oriented attacks are powerful alternatives, as they only alter the security sensitive data inside program to achieve the evil goal, which can bypass most of the current defenses. We look into the practical viability of data-oriented attacks.

A General Method to Build Data-Oriented Attacks. It is difficult to build data-oriented attacks, compared to build control-flow hijacking attacks. We look into them from another perspective, where the original data flows inside the program are split by the memory errors into small pieces and can be re-organized for malicious purpose. With our new perspective, each data-flow piece can perform a specific computation. We develop a novel method, called data-flow stitching, to systematically select and connect these data flows. Note that the program's behavior is the multiplication of all data flows. The method of two data-flow stitching is as follows: We first select one data flow as the source flow and another as the target flow. Then we search along the source flow and the target flow to find the pointers that decide the data-flow direction. At last we connect data flows by manipulating the data-flow pointers. The meaningful new data flow helps attackers mount exploits. We design and implement a framework, called FLOWSTITCH, to systematically search data-oriented attacks from potential solution space. The result shows that FLOWSTITCH is both effective and efficient. Automatic generation of data-oriented attacks is possible.

A Comprehensive Evaluation on the Expressiveness of Data-Oriented Attacks. We consider more extensive usage of data-flow split and re-organization to understand the computational expressiveness of data-oriented attacks. It is commonly understood that the expressiveness of data-oriented attacks is limited, due to the missing ability to divert the control flows. We ask the following question: *what is the real expressive power of data-oriented attacks?* Our answer is that such attacks are Turing-complete. We present a systematic technique called *data-oriented programming* (*DOP*) to con-

struct expressive data-oriented attacks for arbitrary x86 programs. The idea is to identify basic gadgets that perform fundamental calculations and to stitch them together with particular program logics – the dispatcher. The dispatcher selectively connects particular gadgets for malicious computations. We develop a tool to systematically identify gadgets and dispatchers from the vulnerable program. With our tool, we identify a large number of gadgets and dispatchers available from real-world programs. Two of them are confirmed to be able to build Turing-complete attacks. We build 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, respectively. All the attacks work in the presence of ASLR and DEP, demonstrating how the expressiveness offered by DOP significantly empowers the attacker.

Summary of Contributions:

- We study the problem of unexpected memory accesses in privilege-based isolation. We design a novel mechanism to systematically detect DUI vulnerabilities, and to estimate attackers' capability in controlling user memory spaces. Our evaluation on several real-world systems detects severe DUIs in user / kernel isolation and main code / library isolation.
- We conceptualize *data-flow stitching*, a new approach that systematically generates data-oriented attacks, by composing the benign data flows in an application via a memory error. We build a prototype, FLOWSTITCH, to show that constructing data-oriented attacks from common memory errors is feasible, and offer a promising way to bypass many defense mechanisms to control-flow attacks.
- We show that with a single memory error, data-oriented attacks can mount Turing-complete computations with *data-oriented programming (DOP)*. We propose concrete methods to synthesize such expressive attacks. Our evaluation of real world applications shows the prevalence of data-oriented gadgets and dispatchers required by DOP. We build end-to-end data-oriented attacks which work even in the presence of DEP and ASLR.

Chapter 2

Background

In this section, we provide the background of memory errors, including the scope of memory errors discussed in this thesis, existing memory error detection and prevention mechanisms, various exploit methods, and defenses against memory error exploits.

2.1 Memory Error & Detection

In this thesis, we define "memory error" as follows. During the program execution, one memory operations should only access a subset of the full memory space at a specific time . The program's legitimate memory access can be described as a set of legitimate memory accesses. Any memory access that does not belong to the legitimate set is an unexpected memory access. If it shows during program execution, then it is a memory error. There are two types of memory errors: the spatial memory error and the temporal memory error. Both of them are not in the legitimate set: the former one has an incorrect memory space, while the latter one has the incorrect time. The result of the unexpected memory access may vary accordingly. It may cause the program to crash, show some unexpected behaviors, or just work well as expected. Examples of memory errors include buffer overflow [61, 68, 74, 132], buffer underflow [66], format string vulnerability [71, 77, 79, 146], integer overflow [72], use-after-free [78], double free [67], type confusion [64], dangling pointer [73] and so on. Memory error violates the designer's purpose and thus leads the program execution to unexpected behavior.

Memory errors are introduced by the programming mistakes made by developers. Type-unsafe languages, like C and C++, allows developers to manage the program

memory space by themselves. Careless programming is prone to make mistakes, bring memory errors into the code. In fact, as the size of program code base increases and developer's efforts are limited, it is inevitable for a huge program to contain memory errors.

Next, we show the state-of-the-art techniques to detect memory errors and to prevent successful exploits.

2.1.1 Memory Error Detection

To efficiently and effectively detect memory errors, a lot of methods have been developed to detect memory errors. Static detection checks the program source code or binary to detect violation of program static feature. For example, `gcc` compiler detects out-of-bounds access during the compilation. However, static detection cannot find the memory errors only shown during the real execution. Hence we focus on the dynamic detection methods next.

Symbolic Execution. Symbolic execution is commonly used for dynamic vulnerability detection. In symbolic execution, the program is executed with symbols rather than concrete values. Operations on the inputs are represented as an expression of the symbols, naturally providing constraints on possible values of the input after each operation. As a result, symbolic execution [107] has been extensively used in program testing and vulnerability analysis [38, 44, 45, 120, 139]. EXE [45] aims to automatically generate the input that crashes the program. It uses symbolic input to drive the program execution and records all the constraints along the path. By solving the constraints, it can generate concrete inputs that drive down a particular path, including the one triggering memory errors. KLEE [44] is designed to generate program test cases to cover a large portion of the program paths. Catchconv [120] detects memory errors caused by integer conversion errors. Brumley *et al.* [38] proposed an automatic memory error detection method, which detects deviations of different protocol implementations. Darwin [139] uses symbolic execution to find the input that distinguishes the executions of the stable program version and a new version, thus identifying the bugs inside the newer one.

Taint Analysis. Dynamic taint analysis is another technique frequently used to detect vulnerabilities. In taint analysis, each variable is associated with a metadata, recording the source of the data. Attacker-controlled data are usually marked with a special tag in its metadata. Whenever a variable is used to derive other values at runtime, its metadata is propagated to the target data. This enables the analyst to determine the data flow and the attackers' influence. Security analysts predefine a set of program memory access features, and check the program execution with the metadata to detect violation of such features. Many work utilized taint analysis to detect and analyze vulnerabilities [40, 49, 183] and malware [83, 172]. Newsome *et al.* [128] proposed using dynamic taint analysis to find bugs. Brumley *et al.* [40] proposed a method to generate vulnerability signatures, based on the input data set that triggers the vulnerability. HI-CFG [49] introduces a hybrid information- and control-flow graph to generate attacks for vulnerable programs accepting complex inputs. AppSealer [183] automatically generates patches for Android applications to prevent control flow hijacking attacks. Panorama [172] captures the information flow of sensitive data in system level to identify malware and analyze its behavior, including which sensitive data is stolen and where the information is sent.

2.1.2 Memory Safety Enforcement

Memory safety enforcement can prevent all memory errors at the first step. There are two types of memory safety: spatial safety and temporal safety. The spatial safety requires each memory access to be within a valid boundary, while the temporal safety enforces the memory access to be within a valid life-time. The basic idea to enforce spatial memory safety is to add boundary checking log for each memory access inside the protected code. To enforce the temporal safety, the idea is to insert code to check the liveness of the accessed variable. The instrumentation can be performed on program source code, intermediate representation (like LLVM IR), or the binary code.

Spatial Memory Safety Enforcement. Cyclone [105] and CCured [124] introduce a safe type system to the type-unsafe C language. The idea is to use a fat-pointer to record the starting address and end address for each pointer in the program. All memory accesses are checked for out-of-bounds access. This requires source code annotation

and changes the memory space layout. SoftBound [122] uses a dedicated memory space to hold the pointer information. It keeps the memory layout but requires extra memory access to retrieve metadata. Instead of tracking information for each pointer, several work [18, 141] tracks the information for each object. The benefit is that it is not necessary to update the pointer information for each pointer operation, especially pointer operation by uninstrumented code. CRED [141] uses this method to enforce the spatial memory safety. Baggy Bounds Checking [18] aligns each object size to a power of two to implement a fast checking. AddressSanitizer [147] places guard pages between objects to detect out-of-bound memory access. LowFatPtr [82] encodes the length and base information of a pointer inside its address value, and thus provides an efficient implementation.

Temporal Memory Safety Enforcement. To enforcement temporal memory safety, CETS [123] and FreeSentry [174] maintain the pointer life-cycle information inside its metadata region. All memory accesses are checked to make sure the pointer used in the operation pointing to alive objects. Memcheck of Valgrind [126] and Address-Sanitizer [147] instead maintain the life-cycle information for each object. They can detect the memory access to a deallocated object, but cannot find the error that reuse a reallocated object. Cling [17] enforces a type-safe memory space reuse, where the memory space can only be reused for the object with the same type and alignment. It is a replacement of `malloc` function, hence only protecting heap objects.

Enforcing memory safety totally prevents memory errors from the source. However, the introduced performance overhead is usually unacceptable. For example, SoftBound with CETS provides complete memory safety, but suffers from an overhead of 116% on average. To make a balance between memory safety and the performance, there are some proposals that selectively protect security-critical memory space and leave other space unprotected. For example, code pointer integrity (CPI) [109] only protects code pointers, with a dedicated memory location to store code pointers metadata. CPI only has 2.9% overhead on average. However, as the security is sacrificed, CPI can be broken by information leakage attacks [87]

2.2 Exploits and Defenses

Since memory error changes the program's behavior, it provides attackers a opportunity to control program's execution. Note that some memory errors have limited influence on program's execution, so cannot be used attackers to exploit the program.

2.2.1 Memory Error Exploits

Control Flow Hijacking Attacks. Control flow hijacking attacks takes over program's execution path, and redirects it to the malicious path or code region. Memory errors are used to change the indirect control flow transfer targets, like return addresses saved on the stack, and function pointers in memory. The first developed control flow hijacking attack directly injects code into vulnerable program's memory space, and diverts the control flow to the malicious code, called *code injection*. Code injection gives attackers the ability to upload arbitrary code and execute it. However, it requires the memory space to be both writable and executable. As modern systems disable such memory feature, code injection attack requires at least one more step to create the writable and executable memory region. *Code reuse* attacks avoid requiring such memory region and instead reuse the existing code inside the vulnerable program. *Return-to-libc* attack [125] redirects the execution into critical library functions, like system, to mount meaningful attacks. In this case, no new code is introduced into the memory space. Existing functions inside the vulnerable process are reused by attackers. A generalization of return-to-libc attack is the *return-oriented programming (ROP)* [35, 52, 149]. ROP reuses small code fragments with a indirect jump (called gadgets) to achieve Turing-complete execution. Due to the large code base of modern programs, with a high probability attackers can find enough gadgets to mount exploits. A following work [142, 144] shows that building ROP attacks can be simplified and automated. ROP has been transplanted to other architectures [43, 53, 90, 108] and several variations [33, 37].

Data-Oriented Attacks. Instead of diverting the control flow, data-oriented attacks modify the security-sensitive data inside the program to achieve the same goals as control flow hijacking attacks. There are several types of security-sensitive data. The decision making data determines the execution path. For example, the `Safemode` [175]

flag inside IE determines whether it is allowed to load untrusted plug-ins into process without user notification. Altering such data will change the program execution flow, without touching the control data. Program configuration data decides the program settings, some of which are related to the security feature of the program. For example, the *RootDir* in Apache configuration file specifies the root direct of the web server and make files outside secret. Changing this data will affect program's security feature and lead to attacks. Chen *et al.* [56] shows that data-oriented attacks are realistic and can cause damages as severe as control flow hijacking attacks. They manually built data-oriented attacks for several vulnerable programs and demonstrated the possibility. However we can see few data-oriented attacks in real-world since it has constraints on program's control flow.

Exploit methods evolve fast as more advanced defense mechanisms are proposed [159]. We believe new exploit methods will be developed in the future. Nowadays, the common exploit method for control flow hijacking attacks is to use ROP to enable a writable and executable memory location and use code injection to mount attacks. For data-oriented attacks, only Chen *et al.* explored their possibility. There is no deep exploration on them.

Automatic Exploit Generation. Although there are a lot of useful exploit methods, creating a working exploit is a tedious and repeated work. To address this problem, many researchers investigate automatic methods to generate exploits. Heelan [99] discussed algorithms to automatically generate exploits to hijack the control flow for a given vulnerable path. Brumley *et al.* [41] described an automatic exploit generation technique based on program patches. The idea is to identify the difference between the patched and the unpatched binaries, and generate an input to trigger the difference. They proposed that an exploit can be described as a predicate on the program state space. Avgerinos *et al.* [23] inherited and refined this description and discussed Automatic Exploit Generation (AEG). AEG can generate real exploits resulting in a working shell from stack buffer overflow and format string vulnerabilities. The following up work, MergePoint [24], is a symbolic execution tool for large-scale testing of COTS software. Felmetsger *et al.* [88] discussed automatic exploit generation for web applications. Current automatic exploit generation only applies on several types of vulnerabilities. The

scalability of these methods is limited.

2.2.2 Defenses against Exploits

As a second-step defense, researchers develop a lot of mechanisms to prevent attackers from building a working exploit. In this way, even memory errors are triggers by elaborated inputs, attackers cannot mount successful exploits.

Address Randomization. One way to prevent exploits is to force a random value requirement for exploit generation. Address space layout randomization (ASLR) [30] randomizes the memory layout of the protected process. With ASLR, the address of each value (code or data) will be randomized at runtime, and will be different for each concrete execution. Attackers cannot predict the addresses in advance. For example, they cannot guess the target code address (i.e., address of the injected code, address of the library functions, or address of the gadgets) to build control flow hijacking attacks. For data-oriented attacks, attackers have less knowledge of the security-sensitive data, thus likely failed to corrupt it. Currently ASLR implementations deployed on modern systems have low randomization entropy [5, 135, 162]. For example, most of Linux executables are compiled without the "-pie" (position independent executable) option. As a result, the main code of the program has to be loaded into fixed addresses. On Windows system, several important libraries, like msvcr71.dll, hxds.dll, are complied without randomization support [169]. Windows and Linux implement ASLR with different methods: Windows mainly replies on the loading-time program patching, while Linux depends on PIE code. Therefore the resulted performance is also different. Windows ASLR has a larger loading time, but once loaded, the program runs as un-randomized code. But for Linux, there is almost no loading overhead, but each execution of the PIE code will suffer some overhead. There are also proposals that randomize the program in a fine granularity [26, 92, 100, 167] and run-time randomization [20]. These methods improve the security guarantee of ASLR, but are not widely deployed due to the high performance overhead or requirement of reliable binary disassembling.

Data Execution Prevention. To defend code injection attacks, DEP [21] is designed to prevent the data region from being executed. DEP enforces that the executable code

region is not writable, and enforces that the writable data region (like stack, heap) is not executable. The implementation of DEP on Linux system is called NX (non-executable bit). Recent CPU feature EPT (extended page table) by Intel [104] supports to configure a memory page to be readable, writable and executable independently, which provides users a flexible security model. To bypass DEP, attackers tend to launch ROP attacks to set several writable and executable pages. Then they mount the code injection attacks.

Control Flow Integrity. Control flow integrity (CFI) [15] requires the program's execution conform the predefined control flow graph (CFG). By this way, any attack that attempts to divert the program's execution will fail the security checking. Attackers can only change the indirect control flow jump, so that CFI only adds checks for indirect control flow transfers (return addresses or function pointers). There are several ways to implement CFI: one is based on information hiding – only the legitimate target has the secret id and the checking code before indirect transfer will compare the target id with the legitimate one; another way to use the address as the id, and use a dedicated table to maintain the legitimate targets. The first method has a good performance, but suffers from the information leakage attack. The second method is resilient information leakage, but has a high performance overhead. Even with the first method, CFI suffers from a high performance overhead. To make it practical, several work [63, 129, 160, 166, 177, 180, 182] tries to make a balance between the security benefit and the performance overhead, developing coarse-grained CFI. However, recent work [48, 80, 96, 97] shows that with a coarse-grained CFI, attackers still have enough freedom to build control flow hijacking attacks, since the limitations on legitimate targets are relaxed. A recent work, control flow bending [46] shows that even with the deployment of the fine-grained CFI, there are still space for attackers to mount attacks. The reason is that the static CFG allows infeasible dynamic path. If one function is frequently invoked, it can be used as the dispatcher to call infeasible paths.

Data Flow Integrity. Other than control flow integrity, data-flow integrity (DFI) is used to protect the data (control data or non-control data) inside the program. There are two approaches to enforce DFI. One uses dynamic information tracking to identify the type of information flow [158, 170, 173]. The other uses static and/or dynamic analysis

in order to determine the set of legitimate instruction sites where a particular memory can be modified or used [50, 178]. These defenses are not yet practical often requiring large overheads or manual declassification. One particular dynamic tracking approach is outlined by Suh *et al.* [158]. They proposed to add a security tag to the data and to propagate the tag throughout the program's execution. When the program uses certain values (like instructions, load addresses, etc), it checks the tag to determine if the data are spurious or not. If so, an error is raised. This approach is implemented by modifying the hardware architecture. The other approach described by Castro *et al.* [50] involves identifying the set of legitimate sites where data can be modified or used and checking if the modifications are indeed made from this set.

Specific Defenses. There are a lot of defenses that are developed to prevent a particular type of attacks. SafeSEH is used to prevent SEH exploit, where the exception handler saved on stack for Windows executables is used to divert the execution. Shadow stack places the saved return address on another dedicated memory location. It uses the shadow value to check the return address integrity on return, or directly use it to return, Stack smashing protection [164] inserts a canary between a potential buffer and the saved registers and return addresses. To overflow the return address, attackers have to consequently write the stack memory, including the canary, leading to the attack detection. VTint [179] sets the virtual function pointer table used in C++ programs to read-only so that attackers cannot corrupt them to mount control flow hijacking attacks.

Another fundamental method to mitigate the damage of memory error exploits is privilege isolation. It splits the program into many relatively independent components and groups them into several partitions. Each partition is only assigned minimal privileges, which are necessary for that partition to function correctly (a.k.a the least privilege principle). Under this protection, even if some partitions are compromised, attackers can only get the privileges of the compromised partitions. Other partitions are still protected. Privilege separated programs have stronger security guarantee than the monolithic ones.

Process-Based Isolation. As the unit for operation systems to allocate system resources (e.g., memory, opened files), process is naturally selected as the container for

isolated partitions. The privilege-separated program creates several processes at runtime. Only the privileged processes have access to sensitive resources, while the unprivileged processes require information from the privileged ones. The operating system and the underlying hardware help prevent cross boundary access.

There is many work that leans on process-based isolation. Provos *et al.* [138] applied process-based isolation on OpenSSH program and split it into a privileged monitor process and several unprivileged slave processes. They show the isolated program can prevent a set of known attacks, with acceptable performance overhead. Following this direction, Kilpatrick proposed Privman [106], a library that developers can use at development time to protect their code through process-based isolation. Privtrans [42] provides a method to automatically split the existing code into privileged partition and unprivileged ones, with few manual annotations. It identifies all privileged code through program analysis, saving some manual analysis time. Wedge [34] provides a new system primitive, *sthread*, to represent a partition. Compared to the process-based isolation, sthread can afford fine-grained privilege isolation. Within the same memory space, each sthread is assigned privileges to access particular memory regions and files. It based on the dynamic program analysis to figure the privileges associated with each sthread. Process-based isolation is also used to protect the main program from untrusted libraries. Codejail [168] creates a low-privileged process to load each untrusted libraries, like `libpng`. Then the library process periodically commits the updated data to the main process, while the main process selectively updates its own memory.

In contrary to the security benefit, process-based isolation brings a heavy performance overhead. The original communications (e.g., memory accesses, function calls) are changed to inter-process communications (e.g., shared memory, network socket). Program's performance can be significantly slowed down, based on the number of inter-process communications.

Software-Based Isolation. To achieve a better performance, software-based fault isolation (SFI) creates boundaries (fault domain) within the process memory space. SFI enforces that the code inside one fault domain can only access data or transfer control to code within its own fault domain. In this case, the malicious or compromised partition cannot introduce damage to others outside of its fault domain. SFI can be implemented

by checking or sandboxing through code writing. It is the SFI developer's responsibility to enforce the protection at runtime.

SFI has been extensively used and transplanted to many architectures. Wahbe *et al.* [165] first proposed efficient software-based fault isolation on RISC load/store architecture. They designed two methods to conform each partition. One method is to insert checking code for each memory access or control flow transfer instruction. If the target is out of the fault domain, SFI gives warnings. Another way is to always patch the target address to one inside the fault domain, regardless the original target value. They also introduced cross-fault-domain RPC for legitimate communications. Result shows that SFI is a light-weight solution and achieves a balance between security and performance. After the first proposal, several work [86, 114, 152] tried to transplant SFI technique to CISC architecture. The main difference between CISC and RISC system related to SFI is that CISC architecture supports variable-length instructions. In the original SFI proposal, the control flow is allowed to transfer to a validate code entry. Since instruction is 4-byte aligned, this checking is very fast. While in a CISC system instruction length varies from different instruction format, it is slow to confirm that control flow will jump to a valid code entry. To address this problem, PittSFIeld [114] uses binary-writing to align the jailed code. Each valid jump target is relocated to the start of a memory chunk. Only the starting address of a memory chunk is valid target. Then they applied the original SFI on the rewritten binary. Browsers like Chrome extensively use SFI to protect themselves from their various plug-ins. Native Client [171] is a sandboxed execution environment for untrusted x86 native code inside plug-ins. It loads the untrusted native code as a NaCl module, making it portable across operating systems and web browsers. Native Client uses a sandbox to limit the code and data access within the fault domain of NaCl module. To allow communications cross NaCl modules, it provides a narrow set of APIs for the jailed native code. Untrusted plug-ins have to be recompiled with Native Client compiler to support execution inside NaCl module. JIT-NaCl [22] is a patch to Native Client to support just-in-time code and self-modification code. The idea is to introduce the sandboxing code at the JIT code generation and modification time, and enforce the checking at the code execution time.

2.3 The Need for New Systematic Solutions

Existing defense methods can detect and defeat a large portion of known memory errors and exploits. However, the exploit techniques are quickly developing to bypass existing mechanisms. Such changes in memory error exploit methods will bring challenges to current solutions. Instead of building defenses after memory errors and exploits, we need an effective solution to quickly response to new exposed memory errors, and a systematic method to explore potential exploit directions in advance.

Systematic Method to Detect New Memory Errors. Memory errors are caused by original code blocks with memory operations. To detect memory errors, we need a systematic analysis platform to deeply understand the memory access behavior of each block. With this understanding, we can quickly conclude the consequence of the memory and measure the severity when a new memory is exposed. Further, we can detect new memory errors, if we know the new execution environment, when the code is transformed with privilege-separation.

Comprehensive Understanding of New Exploit Vectors. With program isolation or memory error exploits, attackers get the chance to break end-to-end program logic and reorganize them for malicious purposes. They can make the block misbehaved and reconnect it back to the rest of the program in a malicious manner. Code reuse attacks are well-known to public, like return-to-libc attacks and return-oriented programming (ROP). However, as an analogy of ROP in non-control data space, data-oriented attacks can easily bypass existing solutions. We should explore such attacks to understand the attacker's capability with these new exploit vector, like the difficulty to build data-oriented attacks and the expressiveness of such attacks. This understanding will help us build new defense mechanisms to prevent severe damages from malicious exploits in the future.

Chapter 3

Detecting Memory Errors in Privilege-Separated Software

Privilege separation is widely used to secure complex software systems. It divides a monolithic program into several partitions and each partition only has a reduced set of privileges. There are clearly defined boundaries between different partitions and usually they do not share resources by default. Inter-partition communication is only allowed via clearly defined interfaces. In this work, we use *partition* as a general concept of the isolated environment, instead of any concrete mechanism (e.g., process, fault domain). Privilege separation guarantees that even some partitions are malicious or compromised, other partitions and their associated resources are still securely protected. To apply privilege separation on legacy programs, developers need to transform the monolithic legacy programs, either manually or automatically. For example, the OpenSSH server was originally implemented as a monolithic program, where a single vulnerability will expose all critical resources to attackers. To mitigate the threat, part of OpenSSH code without access to high-privileged resources (e.g., password) was separated from other code and isolated as a slave process [138]. In addition, Qmail [29], Postfix [36] and Google Chrome [28, 171] are also designed (or re-designed) with privilege separation.

To facilitate retrofitting legacy code into privilege-separation designs, many solutions have been proposed to partition software and assign each partition a different set of privileges, such as Wedge [34], Privtrans [42], and Privman [106]. The deployed techniques include sandboxing [165, 168, 171] and process-based isolation [34, 42, 106]. When the monolithic code is divided into several partitions, original program behaviors

inside the original code (e.g., function calls or direct memory access) need to be transformed into inter-partition communications (e.g., via socket and shared memory). As a result, program logic that ensures the correctness of program semantics, such as error handling and assertions, may be separated into different partitions and thus fail to enforce the checking. Therefore, additional checking code is usually needed, especially in the high-privileged partitions, to make sure that data from other partitions is valid and legitimate. However, if the transformation fails to include proper checking code in the high-privileged partition, attackers get the chance to use specially-crafted inputs to compromise the high-privileged partitions and carry out privilege escalation attacks. One famous example is the buffer overflow vulnerability, where the high-privileged partition lacks of boundary checking on the buffer access with untrusted data.

Other than the known problems, there are more subtle memory errors in the high-privileged partitions, with which attackers can perform arbitrary memory access inside the high-privileged partition. These memory errors require particular memory access patterns. For example, if a partition uses an input from another untrusted partition as the array index, writing to the array inside the partition is an arbitrary memory access under the influence of the input provider. Attackers can utilize this memory access behavior to modify critical data or retrieve secrets of the partition in a targeted manner.

In this work, we refer to this type of memory access errors by the acronym *DUI (Dereference Under the Influence)*. It stems from the memory access pattern in the vulnerable partition: The address used in memory read or memory write is influenced by malicious data from other partitions. DUI allows cross-partition memory access, violating the goal of privilege-based isolation. Therefore it is necessary to identify them and then handle them specially by programmers. It is worth to note that attackers can corrupt discrete memory locations with DUI exploits, instead of a continuous memory region. This means defenses like stack canary cannot prevent such attacks.

Challenges. DUI exploits can be prevented through sufficient checks on interface inputs. Unfortunately, it is non-trivial to ensure that adequate checking code has been added at correct locations. The checking code in legacy programs is often scattered across many program locations in the monolithic code base, which is split during the privilege-separation transformation, it is necessary to guarantee that each of these program locations are checked correctly. However, to achieve this goal, manual modifica-

tion usually takes a long time to fully understand the requirement of checking operations, while automatic separation methods often miss important checks. Therefore, we need a systematic method to detect such DUI vulnerabilities resulting from privilege-separation transformations.

Our Approach. To address these challenges, we develop an approach, DUI Detector, to automatically and systematically detect code suspicious to DUI exploits in the binary of the high-privileged partitions. Specifically, through dynamic binary program analysis, we identify the suspicious instructions that use data from other partitions as addresses. We choose dynamic analysis to avoid the large number of false positives cased by inaccuracy in static analysis. Then we use symbolic analysis to identify code with the DUI vulnerability and assess the attackers' capabilities in exploiting them. Symbolic analysis makes our approach cover all inputs triggering the same execution path. DUI Detector helps identify concrete code instances that are easily influenced by attackers among a large code base, which should be taken care of by programmers.

We applied our approach on several real-world software systems retrofitted with different types of isolation schemes. DUI Detector successfully detected DUI vulnerabilities inside them. We present four case studies where attackers can perform DUI attacks, in cases of kernel-user isolation and library isolation. Furthermore, our approach reports the attackers' capability to the developers, providing a comprehensive understanding of the vulnerability.

In summary, this work makes the following contributions:

- We study the problem of arbitrary memory access (DUI) in privilege-separation transformations, and identify several general memory access patterns leading to DUI vulnerabilities in binary instruction level.
- We design a novel mechanism to automatically detect DUI vulnerabilities, and to estimate attackers' capability in controlling user memory spaces. It helps developers add sufficient checking.
- We prototype an automated tool and evaluate it on several real-world software. Our tool automatically detects and comprehensively analyzes DUIs in these software programs when they are gone through privilege-separation transformation.

```
1  struct subobj { ... } * p_sub;
2  struct object { ...
3     struct subobj * sub;
4  } * p_obj;
5
6  int main()  {
7     p_obj = create_object();
8     p_sub = create_subobj();
9     p_obj->sub = p_sub;
10 }
11 // create an object instance and return its pointer
12 struct object * create_object() {
13    struct object * ptr = malloc(sizeof(struct object));
14    if (ptr == NULL) { /* error handling */
15    if (isFreed(ptr)) {/* the pointer is freed, error handling */}
16    /* more program specific checks */
17    return ptr;
18 }
19 // create a subobj instance and return its pointer
20 struct subobj * create_subobj() {
21    struct subobj * ptr = malloc(sizeof(struct subobj));
22    if (ptr == NULL) { /* error handling */
23    /* more program specific checks */
24    return ptr;
25 }
```

Code 3.1: Example code to illustrate DUI problem.

3.1 Problem

In this section, we motivate the problem by a concrete example. Then we define the problem of DUI detection and two DUI types: The write DUI and the read DUI.

3.1.1 Motivating Example

We use the example in Code 3.1 to illustrate the memory access problem during the program transformation. In this example, the structure `object` has one pointer of structure `subobj`. Functions `create_object` and `create_subobj` return pointers of new structure instances. The statement on line 9 in function `main` assigns the pointer `p_sub` of a `subobj` instance to the `subobj` field of an `object` instance. Originally function `create_object` and `create_subobj` are in the same partition with the function `main`, and there are checking code inside them to make sure that the return values are correct. During the transformation, these two functions are separated into a low-privileged partition since they are not trusted any more. In this case, the low-privileged partition only works as a memory manager, but does not have access to the memory of main code. Therefore, the returned addresses could be malicious. To protect

```
1  v1 = API_recv();
2  v2 = API_recv();
3  array[v1] = v2;
```

Code 3.2: An example of write DUI

the high-privileged `main` function, we can use memory isolation to prevent the direct memory access from low-privileged code to `main`'s memory, in which case function `create_object` and `create_subobj` just manage `main`'s memory. However, this is inadequate to protect the high-privileged `main`: The statement on line 9 contains a memory error. When the low-privileged partition is malicious, it allows writing a malicious `p_obj` to a memory location `p_sub` inside the protected one. The statement on line 9 is an instance of DUI vulnerability.

3.1.2 Problem Definition

We give the definition of the problem solved in this work.

DUI Detection: Given a partition of a privilege-separated program, we detect whether the partition's memory access behaviors can be influenced by data from its interfaces. In particular, the memory addresses or the data are derived from the interface inputs, giving attackers the ability to read or write to a large range of memory inside the partition.

Attackers use the DUI vulnerability as a memory access service to mount attacks. They specify the address and the data through specially-crafted inputs. DUI vulnerability then finishes the memory operation on behalf of attackers. In real-world programs, the logics used to derive the address from the interface inputs could be complicated, thus subtle and hard to spot. However, the final result is that the attacker can exercise certain levels of influence over the address of the memory operation. It is worthwhile to note that only controlling the memory address is inadequate to corrupt the memory or to steal the sensitive information. Corresponding data flows are necessary to provide the malicious data or send the confidential data out. Based on the direction of the memory access, there are two types of DUI, the write DUI and the read DUI.

Write DUI. We call a memory write operation the *write DUI* if both the memory address and the value to be written in the operation are derived from the interface inputs. Take the code in Code 3.2 as an example. The $API_recv()$ is an interface through

which the code can receive data from other partitions. The memory write operation on line 3 has the address $array + v1$ and the data $v2$ derived from the interface inputs, which allows the input provider to write the selected data to any address in its memory space. We can relax the requirement of the data to be written to a value predictable by attackers. An example is that if $v2$ in Code 3.2 is a constant 0, attackers can use $v2$ to reset important flags, or terminate a C-style string. With the write DUI, attackers can corrupt the memory of the vulnerable program. Not only can they mount control flow hijacking attacks by corrupting code pointers or return addresses, they can also change critical data in memory to mount non-control data attacks [56].

```
1  v1   = API_recv();
2  data = *(base + v1);
3  API_send(data);
```

Code 3.3: An example of read DUI

Read DUI. We call a memory read operation the *read DUI* if the memory address in the operation is derived from the interface inputs and the retrieved data are eventually passed to the output interface of the partition. Consider the example in Code 3.3. $API_recv()$ and $API_send()$ are APIs used by the code to receive data from other partitions and send data to other partitions, respectively. This code snippet retrieves data from a local buffer and sends it out. Since the data retrieving address $base + v1$ is under the control of attackers via the interface input $v1$, attackers can steal sensitive information from the partition. For read DUI, it is insufficient to control the memory read address. The data being read has to reach an output interface for it to complete. In real-world programs, the web client may have secret keys or high-privileged files on the server client. Attackers can use read DUI vulnerability to steal the secret key or file. Another exploit is to leak the randomized address of the loaded modules, leading to bypassing address randomization protections [30, 134].

3.1.3 Memory Access Patterns to Detect DUIs

Although attackers can use various ways to control the memory access, one DUI vulnerability is inevitably represented as attacker-controllable memory address and data in memory access instructions, i.e., the address is derived from the input, and the data

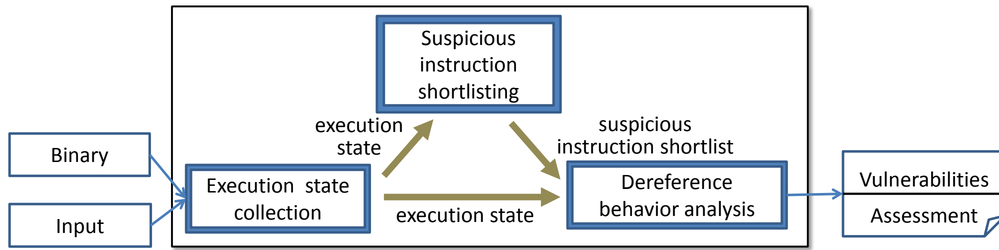

Figure 3.1: Design of the DUI Detector. There are two inputs to the system. One is the program binary, containing the partition to be checked. Another one is a normal input to the program. The output is the list of DUI vulnerabilities and the assessment of attackers' capabilities.

also comes from the input (for the write DUI) or is sent out (for the read DUI). This observation inspires us to use instruction-level memory access patterns to detect DUI vulnerabilities. We summarize the memory access patterns used in DUI exploits below.

- *Write DUI Pattern 1*. The memory write address and the data are derived from the interface inputs. In this case, attackers control both the value and the address in memory write operation.
- *Write DUI Pattern 2*. The memory write address is derived from the interface inputs. The data value is predictable to attackers. Attackers can exploit this code to set the predictable value to any memory address.
- *Read DUI Pattern*. The memory read address is calculated from the interface inputs. The retrieved data are then passed to output interfaces (e.g., via network, file operation or standard printing).

3.2 Design

3.2.1 Overview

Figure 3.1 shows the design of our DUI detection tool, DUI Detector. It takes two inputs: The program binary containing the partition to be checked and a normal input to the program. It detects DUI vulnerabilities during the binary execution for the given input and estimates the capability of attackers obtained by DUI exploits. There are three phases in the process: Execution state collection, suspicious instruction shortlisting and dereference behavior analysis.

Execution State Collection. First we run the program binary in an emulated envi-

ronment with the given input and record all the execution states of the checked partition, including instructions, operands, processor states and memory states. We also log system-level information such as module loading and unloading behaviors.

Suspicious Instruction Shortlisting. From the execution states, our tool identifies instructions potentially vulnerable to DUI exploits. We use data dependency analysis to find the source of the memory address and the data used in memory access instructions. For a memory read operation, we also search forward to check whether the retrieved data is sent to other partitions through output interfaces. If the address is derived from the interface inputs and the data is derived or used in an attacker-controllable manner, we report this memory operation as a suspicious DUI vulnerability.

Dereference Behavior Analysis. Our tool performs symbolic execution to generate the *access formula*, which capture all the constraints from the interface inputs to the suspicious instruction. Then it analyzes the access formula to verify the DUI vulnerability and to assess the capability of attackers in controlling the memory space of the vulnerable partition. DUI Detector reports the verified DUI vulnerabilities together with their severity to developers, helping them to fix the vulnerable code. Next, we introduce the key phases of the DUI Detector.

3.2.2 Suspicious Instruction Shortlisting

From the collected execution states, we use data dependency analysis to track the data flow of the memory address and the data used in memory access instructions. We define the data dependency analysis as follows: Instruction I_0 is *data-dependent* on instruction I_1 if $R(I_0) \cap W(I_1) \neq \emptyset$, where $R(I_i)$ is the set of memory locations or registers read by instruction I_i and $W(I_i)$ is the set of memory locations or registers written by instruction I_i. The methods used to detect DUIs are given below.

To detect write DUIs, we check for the following conditions for each instruction. (1) It is a memory write instruction, i.e., instructions that write the data into memory, like `mov`, `add`, `push` and successful conditional move `cmov`. (2) The source operand is derived from the interface inputs, or predictable to attackers. (3) The address of the destination operand is also derived from the interface inputs.

To detect read DUIs, it is insufficient to check just one single instruction. Other than the actual memory read operation, it is also necessary to identify the data flow

```
1   offset = API_recv(); // API_recv() receives data from others
2   value = API_revc();
3   addr = base + offset;
4   if (addr < MAX_ADDR)
5       * addr = value + 100;
6   value = random(); // random() returns a random value
```

Code 3.4: Pseudo code to demonstrate four constraints in the access formula.

from the read operation to output interfaces, as we discuss in Section 3.1. Hence, we use a two-step approach to identify a read DUI.

1. If it matches the following conditions, we will further check it in step 2: (1) The instruction reads data from memory and saves the data into registers. Instructions reading from registers or without saving the data into registers are ignored. (2) The memory address is derived from the interface inputs.
2. For an instruction selected above, we perform forward slicing on the data flow of the destination operand (the retrieved data). If the data reaches an output interface, we report it as a potential read DUI.

Our tool generates a list of the suspicious instructions potential vulnerable to DUIs. However, strong constraints on the interface inputs could significantly limits the attackers' capability, even making the instruction unexploitable. Hence we need to analyze each suspicious instruction to confirm the vulnerability and assess attackers' capability.

3.2.3 Dereference Behavior Analysis

Given a suspicious instruction identified in the previous step, our tool extracts an access formula to represent the relationship between the interface inputs and the address or data used in the instruction. The access formula captures all the constraints in the execution states with respect to the interface inputs. There are four types of constraints in the access formula as follows. We use the Code 3.4 to demonstrate them.

- **Data-Flow Constraints.** Data-flow constraints describe the arithmetic relations between the address and the data in the DUI instruction and the interface inputs. They are presented as a sequence of arithmetic operations. As in Code 3.4, one data-flow constraint is that `addr` is the sum of `base` and `offset`.

- **Control-Flow Constraints.** Control-flow constraints ensure that the execution follows the same path as the one recorded in the execution states. We only consider the path constraints related to the interface inputs. Other path constraints are out of the attacker's control and are assumed to be satisfied. In Code 3.4, the control-flow constraint to reach line 5 is that `addr` is less than `MAX_ADDR`.
- **Memory Space Constraints.** To reach the suspicious instruction, all the memory accesses should be legitimate. Specifically, the code must have the correct write or read permission of the accessed page. Otherwise, page fault exceptions will be raised and divert the execution path. This constraint limits the attacker's capability since only a subset of memory space is accessible. In line 5 of Code 3.4, the memory write address `addr` has to be writable.
- **Data Life-Cycle Constraints.** To create an effective attack, the malicious data (or retrieved data) must be used within its life-cycle. Otherwise, the suspicious instruction cannot be exploited. For example, if the malicious data written to a selected location is immediately overwritten by a benign value, the attack does not have any effect on the victim partition. To capture this constraint, our tool considers subsequent instructions in order to track the aliveness of the data. In Code 3.4, the variable `value` is updated in line 6 and therefore the malicious value from line 2 has to be used before line 6.

The generated access formula captures all the constraints on the interface inputs to reach the suspicious instruction and continue the execution. There could be over-constraints on the input to reach the instruction, as the execution path could be customized for a concrete input. For example, the control-flow constraint requires the same input length if there exists a loop based on the input length. This problem is out of the scope of this work. We rely on state-of-the-art tools to capture constraints. By assessing the attacker's capability by exploiting the instruction, we can determine if the suspicious instruction is indeed a DUI vulnerability. If so, we report the suspicious instruction and the attackers' capability to developers.

3.2.3.1 Attackers' Capability Assessment

Attackers' capability is represented as the ability to control the address and the value in the memory operation. A larger memory range controllable by attackers indicates a

stronger attackers' capability. However, due to the constraints on the interface inputs, not all the malicious inputs lead to a successful attack. The working inputs form a valid data space, and the attacker's capability is determined by the size of this space. Our tool constructs constraint queries to estimate this space size. Specifically, we assign concrete values or a memory range to the operands of the suspicious instruction, i.e., the address or the data. Then these assignments are added to the access formula as new constraints to form a query. By solving the query using a constraint solver, we get the answer to the following questions: **Q1:** Is there any input making the partition follow the same path to the suspicious instruction when the address or the data in the address have to be a given value? **Q2:** Is there any input making the partition follow the same path to the suspicious instruction when the address or the data have to be within a given range? **Q3:** Is it true that for any address or any data in the given range there is an input making the partition follow the same path to the suspicious instruction? The answer to the question **Q1** indicates attackers' capability on controlling specific addresses. This is useful to build real exploits, for example, writing the ROP gadget address to a function pointer. A negative answer to the question **Q2** helps filter out a memory range from attackers' capability. While a positive answer to the question **Q3** adds the queried range into attackers' capability.

We take several query strategies to efficiently answer these questions. These strategies are based on the bit-pattern analysis and the range analysis [117, 127]. Through these methods, we can estimate the valid memory space controllable by attackers for each of the suspicious instruction.

- **Initial Target Analysis.** We first consider the memory page permission to initialize the memory range. For a memory read operation, the target memory location has to be readable. For a memory write operation, the target memory location has to be writable. Using this method, the queried memory range is limited to the readable or writable memory space.
- **Bit-Pattern Analysis.** Bit pattern analysis uses queries that specify concrete values on particular bits (or all bits) of the target value [117, 127]. An example of the query is whether the last two bits of the address have to be 10. This gives the answer to question **Q1**.
- **Range Analysis.** The range query identifies whether a particular range is valid [127]

or not. If all values in the range are valid, we conclude that the queried range is a valid range. If none is valid, we remove the range from the valid memory space. If only some values are valid, the query solver will give a concrete valid value. We use this value to divide the range to two subranges. Then we use the range query to query both subranges. This query answers question **Q2** and **Q3**.

Finally the report given by DUI Detector includes the identified DUI vulnerabilities, together with the attackers' capability obtained by exploiting such vulnerabilities. It points out all the vulnerable-prone code in the checked partition. This enables security analysts to focus their efforts on a particular portion of the code.

3.3 Implementation

We built a prototype of DUI Detector on a 32-bit Ubuntu 10.04 system by extending the BitBlaze [155] platform. The prototype uses STP [91] as the SMT solver to query the access formula.

3.3.1 Taint Propagation

DUI Detector uses taint analysis to track the data flow of the interface inputs. Data from the interfaces are bound with the taint information of the source. Taint information has two aspects: One aspect is the taint attribute, a flag indicating whether a particular memory byte is tainted or not. Another aspect is the taint record, which contains the sources of the taint attribute. We use TEMU, the dynamic analysis engine of BitBlaze, as the base of taint propagation. However, there are several problems when we use TEMU to build our tool. Next we discuss these problems and present our solutions.

Finer-Gained Taint Record Propagation. Since we need to capture all the execution constraints, the taint propagation has to be accurate to permit the identification of all data sources. The normal taint propagation focuses on taint attribute propagation, and pays less attention on the taint record propagation. For example, for a given instruction, TEMU checks all its operands, and copies the taint records of the first tainted operand to the destination operand. This propagation method loses some taint sources. For example, in the instruction `add %ebx, %eax`, if `eax` and `ebx` are tainted by different data sources, the taint sources of `ebx` get lost. To solve this problem, we in-

stead identify all the tainted source operands and copy all distinct taint records to the destination operand. As a result, the taint records for each operand capture all the data sources used to derive the operand.

1-Level Table Lookup. If a memory read address is tainted, taint analysis has to decide to propagate the taint to the destination operand or not. Table lookup is a method to propagate the taint. However, table lookup results in over-tainting problem, leading to a high false positive. A better tainting method for table lookup is necessary to capture the read DUI and avoid the over-tainting problem. We observe that as more table lookups are performed, attackers likely have increasingly less influence over the destination. As such, we propose the 1-level table lookup: Only propagating the taint for a single level of memory indexing. When the tainted data retrieved from table lookup are used as an index again, we will not propagate the taint. Our implementation uses the most significant bit of the taint attribute to indicate whether it is tainted through table lookup or not. Note that 1-level table lookup will miss attacks that utilize high-level table lookup to corrupt memory locations. However, we believe that the benefit on false positive reduction overweighs the false negative introduced since attacks with high-level lookup are rare in real-world attacks. With 1-level table lookup, our tool captures memory read operations that are strongly controlled by attackers, and skips the weakly-controlled operations.

Taint Propagation for XMM registers. XMM (eXtended Multi-Media) registers are used to speed up the memory operation (e.g., `memcpy`), by joining several 4-byte copies into a single 16-byte operation. TEMU does not support the taint propagation through XMM registers. When tainted data are copied into an XMM register, the taint information gets lost. To support the taint propagation, we extend TEMU to correctly propagate taint to XMM registers and read taint information from XMM registers.

3.3.2 Access Formula Generation

We use VINE [155], the static analysis component of BitBlaze, to generate the formula for memory access from the trace. As discussed in Section 3.2.3, there are four types of constraints affecting memory access. However, VINE only generates two constraints, the data-flow constraint and the control-flow constraint. To bridge this gap, we develop tools to add additional two constraints into the formula. There are two steps to generate

the memory space constraints.

1. In the guest OS, we insert a kernel module to detect the module loading and unloading behaviors. This information is used to estimate the access permission of each memory page, which is required by the memory space constraints. The kernel module sends the update information of the loaded and unloaded module to TEMU. We log such information together with the number of traced instructions when a behavior happens.
2. Using the log file, we can construct the readable and writable memory ranges for each instruction. Specifically, we collect all the modules that are still loaded in the memory for a given instruction. The union of their readable and writable memory ranges is the valid memory space. We add the memory range as a memory space constraint to the access formula.

To generate the data life-cycle constraints for a particular instruction, we search forward from the given instruction in the trace to find the first memory write instruction that overwrites the data at the same address. We call this instruction the update instruction. The data life-cycle of the data starts from the given instruction, and ends at its update instruction.

3.4 Evaluation

We evaluated our approach in the following system: The host OS is a 32bit Ubuntu 10.04 system, running on Openstack Cloud with two 2.4GHz vCPUs and 4GB RAM. The guest OS in TEMU is a 32bit Ubuntu 9.10 system. Next, we present our evaluation results and then discuss the security implication of our findings.

3.4.1 Efficacy

We applied DUI Detector on privilege-isolated programs to detect DUI vulnerabilities in protected partitions. We focus on two particular isolation schemes: The isolation between malicious OS kernels and user space programs [57, 101, 116, 151, 181], and the isolation between malicious libraries and main programs [85, 89, 165, 168, 171]. We ran several programs on Linux system to get the execution trace, which were written to

drive the execution through communications between different partitions. DUI Detector successfully detected read DUI and write DUI vulnerabilities in the protected user space code and the protected program main code. Further, DUI Detector assesses the attackers' capability obtained by exploiting such vulnerabilities. Next we present the details of these DUI vulnerabilities.

3.4.1.1 User-Kernel Isolation

A few proposals remove the OS kernel from the trusted base of the program execution, like hardware-based isolation (e.g., Flicker [115]) and hypervisor-based isolation (e.g., Overshadow [57]). These isolation schemes are designed to protect the sensitive data in user-space programs from the malicious kernel, so the kernel have no direct access to program memory space. Our goal is to detect DUI vulnerabilities inside protected user-space programs that allow the malicious kernel to corrupt programs' private user-space memory.

— Glibc Code Exploitable by `brk`. A write DUI was detected in the `malloc` function, which manages the heap memory for programs. The `malloc` calls the `brk` system call to request a new heap memory and takes the return value as the break value (the upper bound of data segment). Before looking into the detected DUI, we first illustrate the logic in `malloc` handling the return values of `brk`.

```
1   addr1 = brk(0);                    // get the current brk value
2   addr2 = brk(argument);             // request more space
3   *(addr1 + 4) = addr2 - addr1;      // store the size as metadata
```

This code snippet calls `brk` twice to create a heap memory region. The first `brk` call in line 1 is used to get the current break value (saved in $addr1$), which is the start address of the heap. The second `brk` call in line 2 is used to request more memory space and the new break value is stored in $addr2$. The code in line 3 stores the size value (which is $addr2 - addr1$) into the metadata address (which is $addr1 + 4$). One of our tested programs invokes the `malloc` library function to call `brk`. In the execution states, we found two instructions that match the write DUI Pattern 1, as listed in the following assembly code snippet[1].

[1]We use AT&T assembly format throughout the thesis.

```
1  mov %eax, 0x4(%edx)
2  ...
3  mov %eax, 0x4(%edi)
```

For each instruction, both the value and the memory address are derived from the return values of `brk` system calls. By manipulating the system call return values, the malicious kernel can write any value into an arbitrary address in the victim process, even if the process is protected by encryption. We analyzed the capability of attackers and found that only the second instruction is exploitable. For the `mov` instruction on line 1, the data life-cycle constraints show that the value is immediately overwritten by another benign value. For the instruction on line 2, the first return value has to be a multiple of 8. We show the constraints on the return value generated by our tool below, where the $brkn$ is the nth return value. DUI Detector generated the payloads in order to exploit this DUI vulnerability. The generated payloads successfully wrote the given address to the selected stack address.

```
1  condition( brk1%8 == 0 && brk2>brk1 )
2  address = brk1 + 0x2718;
3  data    = (brk2 - brk1 - 0x2718) | 0x1;
```

To explore other paths, we changed the condition to invalidate one of the constraints. The following are two conditions that lead to the write DUIs in other paths. The last one is the scenario of the Iago attack [54]. Note that DUI Detector accurately identified the constraints of Iago attack: The address has to be non-multiple of 8 and the data write to the memory has to be congruent to 1 modulo 8.

```
1  condition(brk1%8 != 0 && brk1<brk2<brk3)
2  address : relies on brk1;
3  data    : relies on brk1 and brk2;
```

— Glibc Code Exploitable by `mmap2`. The second DUI vulnerability in Glibc is in the code handling the `mmap2` system call. The `mmap2` system call on Linux is used to map files or devices into memory in the Linux system. It is widely used by programs to map large files into memory. From the execution trace, we identified a total of 1,653 suspicious instructions matching write DUI patterns. We further reduced them to 302 based on the attackers' capability analysis. Analysis of the remaining 302 instructions

```
1  condition(brk1%8 != 0 && brk1<brk2>brk3)
2  address : relies on brk1;
3  data    : relies on brk1 and brk3;
```

reveals that all of them use values derived from the first 3 `mmap2` return values. Here we show the very first instruction among them. This is a write DUI, where the memory address and the data are derived from the first and the third `mmap2` return values.

```
1  mov %eax, 0x1ac(%edi)
```

Using the queries we discuss in Section 3.2.3, we identified the valid memory space which the attacker can write values to. For a stack memory range over `0x0BFFF000` to `0x0BFFF2FF`, we found that the attacker has no control over addresses whose last four bits are 1100 or 0100.

— `cat` Exploitable by `read` and `write`. The UNIX utility program `cat` reads data from the given files, concatenates the content and writes them out to the standard output file. This behavior results in consecutive file read and write operations. The `cat` program we used is a derivative of the BSD `cat` program[2]. We identified read DUIs in the `cat` code, which can be exploited by malicious kernel to steal program's private information. To illustrate the read DUI, we present the pseudo code below.

```
1  nr = read(rfd, buf, size);
2  for(off = 0; nr; nr -= nw, off += nw)
3  {
4    nw = write(wfd, buf + off, nr);
5    if (nw == 0 || nw == -1)
6      goto error;
7  }
```

The loop condition nr is fully controlled by the malicious kernel. First, it is initialized by the return value of the `read` system call on line 1. For each loop, it is updated by the return value `nw` of the `write` call on line 2. `nw` is also used to advance the buffer for the next `write` call. When the kernel is changed to be untrusted, isolation mechanisms use deep copy to duplicate all system call parameters to a shared memory between kernel and process [151]. In this case, by manipulating the return value `nw`, the malicious kernel drives the process to copy its private data into shared memory

[2] `http://www.opensource.apple.com/source/text_cmds/text_cmds-87/cat/cat.c`

space. With further capability analysis, we find that the attacker has full control over the value, i.e., the attacker is able to access any memory with the values ranging from `0x00000000` to `0xFFFFFFFF`.

3.4.1.2 Library Isolation

Dynamic shared libraries are linked to software process at the runtime. Since the dynamic library lives in the same memory space with the program's main code, any vulnerability in the library is inherited by the program. Memory separation designs [165, 168, 171] provide transparent memory isolation between the main code and libraries. The goal is to prevent the untrusted libraries from directly accessing the main memory. However, attackers can still leverage the DUIs in the main code to indirectly access the main memory. Some work [168] requires sanitization of the untrusted input or return value. However, they leave the identification of sanitization work to the developers. DUI Detector provides a complementary tool to such isolation schemes.

— Programs Using `libsdl`. The Simple DirectMedia Layer (SDL) library provides programming interfaces to access low lever hardware, like keyboard, screen, audio and so on. The main program requests an SDL object and performs operation through the SDL object. For example, the main program can request a screen object, and then invoke the screen object methods to set display attributes, like colors and fonts. When the library isolation technique Codejail [168] is used, the SDL library code cannot directly access the memory of the main code. A monitor module will selectively commit the memory changes from the library to the main code. However, only isolating memory with Codejail cannot prevent the memory access from the library to the main memory through DUIs in the main code – sanitization code is necessary. We write a simple program that requests a screen object from the SDL library and then sets the color attribute. The pseudo code of the simple program is shown below.

```
1  screen = SDL_SetVideoMode(...);  // get framebuffer surface
2  color = SDL_MapRGB(...);         // get a pixel value
3  pixmem16 = screen->pixels + x + y * pixelsperline ;
4                                   // get pixel address
5  *pixmem16 = color;               // set the color
```

Our tool detected the write DUI in the main code (on line 5) of this simple program.

Table 3.1: Performance of the DUI Detector. T1 is the time for trace generation; T2 is the time to get the access formula; T3 is the time to solve the formula. "Inst. #" is the number of executed instruction, while "Infl. #" is the number of tainted instructions. All times are measured in second.

Trusted Part	**Untrusted Part**	**APIs**	**DUI**	**Inst. #**	**Infl. #**	**T1**	**T2**	**T3**
user space	Linux kernel	brk	write	168,089	103	21.79	1.70	0.18
user space	Linux kernel	mmap2	write	167,644	69,486	21.19	2.94	3.11
cat code	Linux kernel	read, write	read	2,288,914	684	104.76	16.58	0.16
main code	SDL library	SDL APIs	write	100,424,507	68	7574.23	1.52	0.10

A malicious SDL library can exploit this DUI vulnerability to corrupt any memory location of the main code, even if the main program is protected by memory isolation schemes. Using attackers' capability estimation, our tool reports that there is no limitation on the address or value, which means that attackers have full control of the main code memory through the DUI exploit.

3.4.2 Performance

Table 3.1 shows the performance details of each experiment conducted using our tool. We can see that our tool is able to analyze and detect a DUI vulnerability in a few minutes. The time required for the generation of the trace is largely dependent on the number of instructions that are generated and logged in the trace. On the other hand, the amount of time required for the generation of the STP formula is very small. For the STP formula solving, the time required highly varies due to its dependence on the query inputs, formula and how quickly the STP solver can obtain a solution for us.

3.4.3 Security Implications

Our tool detected DUI vulnerabilities in different program transformation scenarios, including untrusted kernel isolation and untrusted libraries isolation. In this part, we discuss the security implications of our findings.

- *Simple memory isolation is inadequate to prevent unauthorized memory access.* Although many designs aim to prevent the malicious partition from accessing the protected memory, our result shows that simple memory isolation cannot completely stop the unauthorized memory access. DUI vulnerabilities inside the pro-

tected partition still allow other partitions to access arbitrary protected memory.

- *API-review is necessary to provide a secure environment.* Since DUI vulnerabilities can be leveraged to mount attacks through interfaces, developers need to pay special attention to the checking code on interface inputs when the legacy code is retrofitted into a memory isolation model. More specifically, there is a need to review the interfaces between trusted and untrusted partitions.

3.5 Discussion

In this section we present the limitation of our work and discuss the possible defense against the DUI exploits.

Legacy Code Reuse. Legacy code is usually written without least-privilege principle in mind. Therefore it is prone to inadequate sanitization when adopting a new threat model. For example, the user space code is written with implicit trust on kernel services, and thus we can expect attacks like Iago. In fact, the correct strategy is to rewrite the user space code with sufficient checking when adopting a new threat model. However, it is a tedious work due to the large code base and the large number of applications. Instead, developers prefer to reuse existing legacy code, as much as possible, to save development effort. Our method provides a way to achieve both security and efficiency, shortening the development cycle and at the same time providing security to split code. Further, even for the code rewriting, our tool can be used to figure out the critical code sections that require developers to write careful checking code.

Code Coverage. Since our method analyzes the trace dynamically generated from one concrete input, it covers the single code path that gets executed during trace generation. The symbolic execution guarantees that all inputs that triggers the same code path are covered. It is possible for the program to have other DUI vulnerabilities in other paths. To cover such code paths to detect other vulnerabilities, we employ an iterative process. Specifically, after the analysis for one execution path, we invalidate the path condition in the control-flow constraints and ask the solver to provide an input that satisfies the invalidated condition [93]. The given input makes the program follow a new code path. The same analysis is performed on it and this process is repeated until

no additional new path can be generated. However, this may lead to the path explosion problem [19]. To mitigate the problem, we only invalidate the conditional branches that are affected by untrusted input to generate the new path. In our evaluation, we limit the exploration of different paths to five. Even with this small number, we still find several DUI vulnerabilities. Further, existing methods to mitigate path explosion, like [112, 157] can also be applied here.

Defense. There are two potential ways to prevent DUI exploits. One is to patch the program with proper checking. Once a DUI vulnerability has been identified, developers can mitigate the consequences of the vulnerability by introducing proper checks to the vulnerable code. Different checks should be used accordingly based on the type of the interface inputs. For the Glibc `brk` attack, where the interface inputs are addresses, the sanitization code needs to make sure that the returned address either equals to the requested one or points to a newly allocated memory region [101]. For operation counters (e.g., the return value of the `read` system call), sanitization code should perform strict checks on the length, like comparing it with the file size or the buffer size. Another method is to use data-flow tracking to dynamically detect the DUI code. This method does not require DUI detection before real execution. It can detect DUI exploit at runtime. However, based on existing knowledge, the performance overhead would be very large, which is usually unacceptable.

3.6 Related Work

Vulnerability Detection. Symbolic execution and dynamic taint analysis are two techniques that are commonly used for vulnerability detection. In symbolic execution, the program is executed with symbols rather than concrete values. Operations on the inputs are represented as an expression of the symbols, naturally providing constraints on possible values of the input after each operation. As a result, symbolic execution [107] has been extensively used in program testing and vulnerability analysis [38,44,45,120,139]. Dynamic taint analysis is another technique frequently used to detect vulnerabilities. In taint analysis, attacker-controlled inputs are usually marked with a tag. This tag is propagated whenever the data is derived from the input. This enables the analyst to determine the data flow and the attackers' influence. A series of work has utilized taint

analysis to detect and analyze vulnerabilities [40, 49, 183] and malware [83, 172]. Newsome *et al.* [128] proposed using dynamic taint analysis to find bugs. In these methods, attacks are detected when the tainted data are used in a dangerous way, like jump address or system call parameters. Our approach differs in application of these techniques. In order to detect DUI vulnerabilities, our focus is on detecting certain access pattern while at the same time considering implicit constraints such as memory constraints. As such, our approach aims to obtain a better understanding of the vulnerability in addition to detection.

Privilege Separation in Software Systems. Privilege separation is a way to realize the principle of least privilege in software designs. It is often achieved by using memory isolation to protect resources of high-privileged partitions from low-privileged ones. For examples, Provos [138] retrofitted OpenSSH with a privilege-separated design and other methods [34, 42, 106] automatically separate and isolate components within monolithic legacy programs. Other security solutions proposed new threat models. For example, some [57, 110, 115] treat the kernel as potentially untrusted and remove it from the trusted computing base. However, the work [54, 137] shows that just isolating the components is insufficient as attackers might be able to leverage on poorly designed legacy interfaces to compromise the isolated components. Our solution complements this work with a systematic method to detect DUI vulnerabilities when adopting new isolation schemes.

3.7 Summary

In this work, we present a systematic solution to detect arbitrary memory access vulnerability in binary programs during privilege-based transformation. Our approach builds access formula for a binary using program analysis techniques. The formula is then utilized to detect the memory access patterns that can be leveraged by attackers to perform arbitrary memory accesses. Detailed analysis is also performed to assess the capability of attackers using such vulnerabilities. We demonstrate the effectiveness and accuracy of our approach in the evaluation, where we present four case studies of DUI vulnerabilities in programs utilizing isolation schemes. Finally, we provide the security implications based on the results of the evaluation.

Chapter 4

Exploiting Memory Errors with Data-Flow Stitching

In previous section we detect memory errors during program transformation, where the execution environment of split components changes. These memory errors (in fact all memory errors) give malicious parties, like the untrusted kernel, ability to manipulate the behavior of the vulnerable programs to some extent. With such ability, attackers always seek to build exploits to execute arbitrary malicious code, which gives them the ultimate freedom in perpetrating damage with the victim program's privileges. Such attacks typically hijack the program's control flow by exploiting memory errors. However, control-oriented attacks, including code-injection and code-reuse attacks, can be thwarted by efficient defense mechanisms such as control-flow integrity (CFI) [15, 180, 182], data execution prevention (DEP) [21], and address space layout randomization (ASLR) [30, 134]. Recently, these defenses have become practical and are gaining universal adoption in commodity operating systems and compilers [121, 160], making control-oriented attacks increasingly difficult.

However, control-oriented attacks are not the only malicious consequence of memory errors. Memory errors also enable attacks through corrupting non-control data – a well-known result from Chen *et al.* [56]. We refer to the general class of non-control data attacks as *data-oriented attacks*, which allow attackers to tamper with the program's data or cause the program to disclose secret data inadvertently. Several recent high-profile vulnerabilities have highlighted the menace of these attacks. In a recent ex-

ploit on Internet Explorer (IE) 10, it has been shown that changing a single byte – specifically the *Safemode* flag – is sufficient to run arbitrary code in the IE process [119]. The Heartbleed vulnerability is another example wherein sensitive data in an SSL-enabled server could be leaked without hijacking the control-flow of the application [13].

If data-oriented attacks can be constructed such that the exploited program follows a legitimate control flow path, they offer a realistic attack mechanism to cause damage even in the presence of state-of-the-art control-flow defenses, such as DEP, CFI and ASLR. However, although data-oriented attacks are conceptually understood, most of the known attacks are straightforward corruption of non-control data. No systematic methods to identify and construct these exploits from memory errors have been developed yet to demonstrate the power of data-oriented attacks. In this work, we study systematic techniques for automatically constructing data-oriented exploits from given memory corruption flaws.

Based on a new concept called *data-flow stitching*, we develop a novel solution that enables us to systematize the understanding and construction of data-oriented attacks. The intuition behind this approach is that non-control data is often far more abundant than control data in a program's memory space; as a result, there exists an opportunity to reuse existing data-flow patterns in the program to do the attacker's bidding. The main idea of data-flow stitching is to "stitch" existing data-flow paths in the program to form new (unintended) data-flow paths via exploiting memory errors. Data-flow stitching can thus connect two or more data-flow paths that are disjoint in the benign execution of the program. Such a stitched execution, for instance, allows the attacker to write out a secret value (e.g., cryptographic keys) to the program's public output, which otherwise would only be used in private operations of the application.

Problem. Our goal is to check whether a program is exploitable via data-oriented attacks, and if so, to automatically generate working data-oriented exploits. We aim to develop an exploit generation toolkit that can be used in conjunction with a dynamic bug-finding tool. Specifically, from an input that triggers a memory corruption bug, with the knowledge of the program, our toolkit constructs data-oriented exploits.

Compared to control-oriented attacks, data-oriented attacks are more difficult to carry out, since attackers cannot run malicious code of their choice even after the at-

tack. Though non-control data is abundant in a typical program's memory space, due to the large range of possibilities for memory corruption and their subtle influence on program memory states, identifying how to corrupt memory values for a successful exploit is difficult. The main challenge lies in searching through the large space of memory state configurations, such that the attack exhibits an unintended data consequence, such as information disclosure or privilege escalation. An additional practical challenge is that defenses such as ASLR randomize addresses, making it even harder since absolute address values cannot be used in exploit payloads.

Our Approach. In this work, we develop a novel solution to construct data-oriented exploits through data-flow stitching. Our approach consists of a variety of techniques that stitch data flows in a much more efficient manner compared to manual analysis or brute-force searching. We develop ways to prioritize the searching for data-flow stitches that require a single new edge or a small number of new edges in the new data-flow path. We also develop techniques to address the challenges caused by limited knowledge of memory layout. To further prune the search space, we model the path constraints along the new data-flow path using symbolic execution, and check its feasibility using SMT solvers. This can efficiently prune out memory corruptions that cause the attacker to lose control over the application's execution, like triggering exceptions, failing on compiler-inserted runtime checks, or causing the program to abort abruptly. By addressing these challenges, a data-oriented attack that causes unintended behavior can be constructed, without violating control-flow requirements in the victim program.

We build a tool called FLOWSTITCH embodying these techniques, which operates directly on x86 binaries. FLOWSTITCH takes as input a vulnerable program with a memory error, an input that exploits the memory error, as well as benign inputs to that program. It employs dynamic binary analysis to construct an information-flow graph, and efficiently searches for data flows to be stitched. FLOWSTITCH outputs a working data-oriented exploit that either leaks or tampers with sensitive data.

Results. We show that automatic data-oriented exploit generation is feasible. In our evaluation, we find that multiple data-flow exploits can often be constructed from a single vulnerability. We test FLOWSTITCH on eight real-world vulnerable applications,

and FLOWSTITCH automatically constructs 19 data-oriented exploits from eight applications, 16 of which are previously unknown to be feasible from known memory errors. All constructed exploits violate memory safety, but completely respect fine-grained CFI constraints. That is, they create no new edges in the static control-flow graph. All the attacks work with the DEP protection turned on, and 10 exploits (out 19) work even when ASLR is enabled. The majority of known data-oriented attacks (c.f. Chen *et. al.* [56], Heartbleed [13], IE-Safemode [119]) are straightforward non-control data corruption attacks, requiring at most one data-flow edge. In contrast, seven exploits we have constructed are only feasible with the addition of multiple data-flow edges in the data-flow graph, showing the efficacy of our automatic construction techniques.

Contributions. This work makes the following contributions:

- We conceptualize *data-flow stitching* and develop a new approach that systematizes the construction of data-oriented attacks, by composing the benign data flows in an application via a memory error.
- We build a prototype of our approach in an automatic data-oriented attack generation tool called FLOWSTITCH. FLOWSTITCH operates directly on Windows and Linux x86 binaries.
- We show that constructing data-oriented attacks from common memory errors is feasible, and offer a promising way to bypass many defense mechanisms to control-flow attacks. Specifically, we show that 16 previously unknown and 3 known data-oriented attacks are feasible from 8 vulnerabilities. All our 19 constructed attacks bypass DEP and the CFI checks, and 10 of them bypass ASLR.

4.1 Problem Definition

4.1.1 Motivating Example

The following example shown in Code 4.1 is modeled after a web server. It loads the web site's private key from a file, and uses it to establish an HTTPS connection with the client. After receiving the input — a file name, the code sanitizes the input by invoking `checkInput()` (on line 10). The code then retrieves the file content

```
1  int server() {
2    char *userInput, *reqFile;
3    char *privKey, *result, output[BUFSIZE];
4    char fullPath[BUFSIZE] = "/path/to/root/";
5
6    privKey = loadPrivKey("/path/to/privKey");
7    /* HTTPS connection using privKey */
8    GetConnection(privKey, ...);
9    userInput = read_socket();
10   if (checkInput(userInput)) {
11     /* user input OK, parse request */
12     reqFile = getFileName(userInput);
13     /* stack buffer overflow */
14     strcat(fullPath, reqFile);
15     result = retrieve(fullPath);
16     sprintf(output,"%s:%s",reqFile,result);
17     sendOut(output);
18   }
19 }
```

Code 4.1: Vulnerable code snippet. String concatenation on line 14 introduces a stack buffer overflow vulnerability.

and sends the content and the file name back to the client. There is a stack buffer overflow vulnerability on line 14, through which the client can corrupt the stack memory immediately after the `fullPath` buffer.

However, there is no obvious security-sensitive non-control data [56] on the stack of the vulnerable function. To create a data-oriented attack, we analyze the data flow patterns in the program's execution under a benign input, which contains at least two data flows: the flow involving the sensitive private key pointed to by the pointer named `privKey`, and the flow involving the input file name pointed by the pointer named `reqFile`, which is written out to the program's public outputs. Note that in the benign run, these two data flows do not intersect — that is, they have no shared variables or direct data dependence between them, but we can corrupt memory in such a way that the secret private key gets written out to the public output. Specifically, buffer overflow of the `fullPath` can make `reqFile` point to the private key. This forces the program to copy the private key to the output buffer in the `sprintf` function on line 16, and then the program sends the output buffer to the client on line 17. Note that the attack alters no control data, and executes the same execution path as the benign run.

This example illustrates the idea of *data-flow stitching*, an exploit mechanism to manipulate the benign data flows in a program execution without changing its control flow. Though it is not difficult to manually analyze this simplified example to construct a

data-oriented attack, real-world programs are much more complex and often available in binary-only form. Constructing data-oriented attacks for such programs is a challenging task we tackle in this work.

4.1.2 Objectives & Threat Model

In this work, we aim to develop techniques to automatically construct data-oriented attacks by stitching data flows. The generated data-oriented attacks result in the following consequences:

G1: Information Disclosure. The attacks leak sensitive data to attackers. Specifically, we target the following sources of security-sensitive data:

- **Passwords and private keys**. Leaking passwords and private keys help bypass authentication and break secure channels established by encryption techniques.
- **Randomized values**. Several memory protection defenses utilize randomized values generated by the program at runtime, such as stack canaries, CFI-enforcing tags, and randomized addresses. Disclosure of such information allows attackers bypass randomization-based defenses.

G2: Privilege Escalation. The attacks grant attackers the access to privileged application resources. Specifically, we focus on the following kinds of program data:

- **System call parameters**. System calls are used for high-privilege operations, like `setuid()`. Corrupting system call parameters can lead to privilege escalation.
- **Configuration settings**. Program configuration data, especially for server programs (e.g., data loaded from `httpd.conf` for Apache servers) specifies critical information, such as the user's permission and the root directory of the web server. Corrupting such data directly escalates privilege.

Threat Model. We assume the execution environment has deployed defense mechanisms against control-flow hijacking attacks, such as fine-grained CFI [15, 129], non-executable data [21] and state-of-the-art implementation of ASLR. Attackers cannot mount control flow hijacking attacks. All non-deterministic system generated values, e.g., stack-canaries or CFI tags, are assumed to be secret and unknown to attackers.

4.1.3 Problem Definition

To systematically construct data-oriented exploits, we introduce a new abstraction called the *two-dimensional data-flow graph (2D-DFG)*, which represents the flows of data in a given program execution in two dimensions: memory addresses and execution time. A 2D-DFG is a directed graph, represented as $G = \{V, E\}$, where V is the set of vertices, and E is the set of edges. A vertex in V is a variable instance, i.e., a point in the two dimensional address-time space, denoted as (a, t), where a is the address of the variable, and t is a representation of the execution time when the variable instance is created. The address includes both memory addresses and register names[1], and the execution time is represented as an instruction counter in the execution trace of the program. An edge (v', v) from vertex v' to vertex v denotes a data dependency created during the execution, i.e., the value of v or the address of v is derived from the value of v'. Therefore, the 2D-DFG also embodies the "points to" relation between pointer variables and pointed variables. Each vertex v has a `value` property, denoted as $v.value$.

A new vertex $v = (a, t)$ is created if an instruction writes to address a at the execution time t. A new data edge (v′, v) is created if an instruction takes v' as the source operand and takes v as a destination operand. A new address edge (v′, v) is created if an instruction takes v′ as the address of one operand v. Therefore, an instruction may create several vertices at a given point in execution if it changes more than one variables, for instance in the loop-prefixed instructions (e.g., `repmov`). Note that the 2D-DFG is a representation of the direct data dependencies created in a program execution under a concrete input, not the static data-flow graph often used in static analysis. Figure 4.1 shows a 2D-DFG of Code 4.1.

We define the core problem of *data-flow stitching* as follows. For a program with a memory error, we take the following parameters as the input: a 2D-DFG G from a benign execution of the program, a memory error influence I, and two vertices v_S (source) and v_T (target). In our example, v_S is the private key, shown as (a1[2], 6) in Figure 4.1 and v_T is the public output buffer, shown as (output, 16) in Figure 4.1. Our goal is to generate an exploit that exhibits a new 2D-DFG $G' = \{V', E'\}$, where V' and E' result from the memory error exploit, and that G' contains data-flow paths from

[1] We treat the register name as a special memory address.

[2] a1 is the private key buffer address, a concrete value.

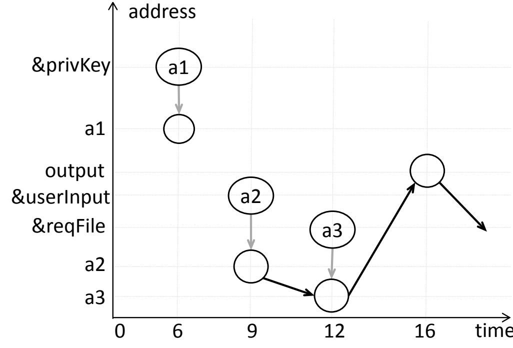

Figure 4.1: 2D-DFG of a concrete execution of Code 4.1. Black edges are data edges, while grey edges are address edges. For clarity, vertices do not conform to the order on address-axis (this applies to all figures). Time is represented by line numbers. $a1$ is the address of the private key. $a2$ is the address of user input. $a3$ is the address of file name.

v_S to v_T. Let $\overline{E} = E'{-}E$ be the edge-set difference and $\overline{V} = V' - V$ be the vertex-set difference. Then, $\overline{E}$ is the set of new edges we need to generate to get E' from E.

The memory error influence I is the set of memory locations which can be written to by the memory error, represented as a set of vertices. Therefore, we must select $\overline{V}$ to be a subset of vertices in I. To achieve **G1** we consider variables carrying program secrets as source vertices and variables written to public outputs as target vertices. In the development of attacks for **G2**, source vertices are attacker-controlled variables and target vertices are security-critical variables such as system call parameters. A successful data-oriented attack should additionally satisfy the following critical requirements:

- **R1.** Exploit satisfies the program path constraints to reach the memory error, create new edges and continue the execution to reach the instruction creating v_T.
- **R2.** The instructions executed in the exploit must conform to the program's static control flow graph.

4.1.4 Key Technique & Challenges

The key idea in data-flow stitching is to efficiently search for the new data-flow edge set $\overline{E}$ to add in G' such that it creates new data-flow paths from v_S to v_T. For each edge $(x, y) \in \overline{E}$, x is data-dependent on v_S and v_T is data-dependent on y. We denote the sub-graph of G containing all the vertices that are data-dependent on v_S as the source flow. We also denote the sub-graph of G containing all the vertices that v_T is data-dependent on as the target flow. For each vertex pair (x, y), where x is in the source

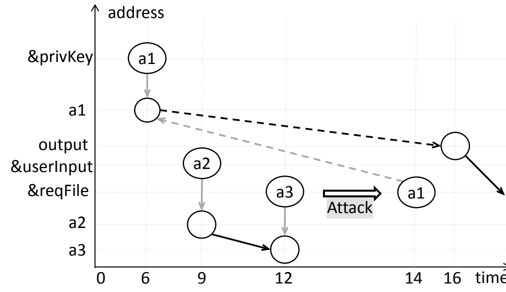

Figure 4.2: A data-oriented attack of Code 4.1. This attack connects flow of the private key and flow of the file name, with the new edges (dashed lines). “Attack” arrow shows the corruption.

flow and y is in the target flow, we check whether (x, y) is a feasible edge of $\overline{E}$ resulting from the inclusion of vertices from I. The vertices x and y may either be contained in I directly, or be connected via a sequence of edges by corruption of their pointers which are in I. If we change the address to which x is written, or change the address from which y is read, the value of x will flow to y. If so, we call (x, y) the *stitch edge*, x the *stitch source*, and y the *stitch target*. For example, in Figure 4.2, we change the pointer (which is in I) of the file name from $a3$ (address of the file name) to $a1$ (address of the private key). Then the flow of the private key and the flow of the file name are stitched, as we discuss in Section 4.1.1. In finding data-flow stitching in the 2D-DFG, we face the following challenges:

- **C1. Large search space for stitching.** A 2D-DFG from a real-world program has many data flows and a large number of vertices. For example, there are 776 source vertices and 56 target vertices in one of `SSHD` attacks. Therefore, the search space to find a feasible path is large, for we often need heavy analysis to connect each pair of vertices.
- **C2. Limited knowledge of memory layout.** Most of the modern operating systems have enabled ASLR by default. The base addresses of data memory regions, like the stack and the heap, are randomized and thus are difficult to predict.

The 2D-DFG captures only the data dependencies in the execution, abstracting away control dependence and any conditional constraints the program imposes along the execution path. To satisfy the requirements **R1** and **R2** completely, the following challenge must be addressed:

- **C3. Complex program path constraints.** A successful data-oriented attack

```
1   struct passwd { uid_t pw_uid; ... } *pw;
2   ...
3   int uid = getuid();
4   pw->pw_uid = uid;
5   printf(...);      //format string error
6   ...
7   seteuid(0);               //set root uid
8   ...
9   seteuid(pw->pw_uid);   //set normal uid
10  ...
```

Code 4.2: Code snippet of `wu-ftpd`, setting uid back to process user id.

causes the victim program execute to the memory error, create stitch edges, and continue without crashing. This requires the input to satisfy all path constraints, respect the control flow integrity constraints, and avoid invalid memory accesses.

4.2 Data-Flow Stitching

Data-oriented exploits can manipulate data-flow paths in a number of different ways to stitch the source and target vertices. The solution space can be categorized based on the number of new edges added by the exploit. The simplest case of data-oriented exploits is when the exploit adds a single new edge. More complex exploits that use a sequence of corrupted values can be crafted when a single-edge stitch is infeasible. We discuss these cases to solve challenge **C1** in Section 4.2.1 and 4.2.2. To overcome the challenge **C2**, we develop two methods to make data-oriented attacks work even when ASLR is deployed, discussed in Section 4.2.3. For each stitch candidate, we consider the path constraints and CFI requirement (**C3**) to generate input that trigger the stitch edge in Section 4.3.4.

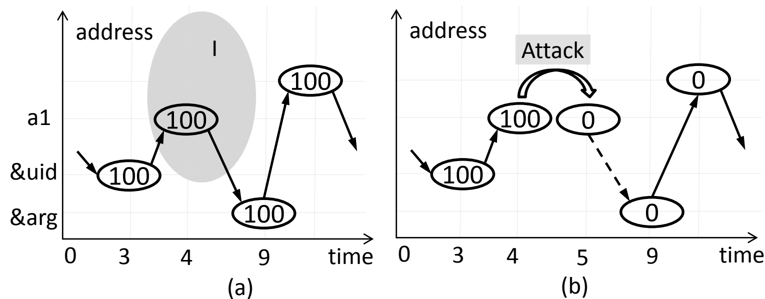

Figure 4.3: Target flow in the single-edge stitch of `wu-ftpd`. &arg is the stack address of `setuid`'s argument. (a) is the original target flow, where the `pw->pwd_uid` has value 100 and address $a1$. Grey area is the memory influence I. The stitching attack changes the value at address $a1$ to 0 in (b).

StitchAlgo-1: Single-edge Stitch

```
Input: G: benign 2D-DFG,  I: memory influence,
       v_T: target vertex,   cp: memory error vertex,
       X: value to be in V_T.value (requirement for stitch edge)
Output: Ē: stitch edge candidate set
1 Ē = ∅
2 TDFlow = dataSubgraph(G, v_T)                      /* only consider data edges */
3 foreach v ∈ V(TDFlow) do
4     if isRegister(v) then
5         continue                                    /* Skip registers */
6     if ∃ (v, v′) ∈ E(TDFlow): ∃ t : v.time < t < v′.time ∧ (v.address, t) ∈ I then
7         Ē = Ē ∪ {(cp, v)}                          /* Stitch edge candidate */
/* dataSubGraph(G, v): the largest direct-connected subgraph of G containing v.   */
/* "direct-connect" means connected by data edges only, excluding address edges.   */
/* V(G): the set of the nodes inside G. E(G): the set of the edges inside G.        */
/* isRegister(v): true if v is a register node, false otherwise.                    */
/* The same meaning applies to StitchAlgo-2.                                         */
```

4.2.1 Basic Stitching Technique

A basic data-oriented exploit adds one edge in the new edge set $\overline{E}$ to connect v_S with v_T. We call this case a *single-edge stitch*. For instance, attackers can create a single new vertex at the memory corruption point by overwriting a security-critical data value, causing escalation of privileges. Most of the previously known data-oriented attacks are cases of single-edge stitches, including attacks studied by Chen *et al.* [56] and the IE Safemode attack [119]. We use the example of a vulnerable web server `wu-ftpd`, shown in Code 4.2, which was used by Chen *et al.* to explain non-control data attacks [56]. In this exploit, the attackers utilizes a format string vulnerability (on line 5) to overwrite the security-critical `pw->pw_uid` with root user's id. The subsequent `setuid` call on line 9, which is intended to drop the process privileges, instead makes the program retain its root user privileges. Figure 4.3 (a) and Figure 4.3 (b) show the 2D-DFG for the execution of the vulnerable code fragment under a benign and the exploit payload respectively. Numbers on time-axis are the line numbers in Code 4.2. The exploit aims to introduce a single edge to write a zero value from the network input to the memory allocated to the `pw->pw_id`. Note that the exploit is a valid path in the static CFG.

Search for Single-Edge Stitch. Instead of brute-forcing all vertices in the target flow for a stitch edge, we propose a method that utilizes the influence set I of the memory error to prune the search space. The influence set I contains vertices that can be cor-

rupted by the memory error, like the grey area shown in Figure 4.3. For vertices in the target flow, attackers can only affect those in the intersection of the target flow and the influence I. Other vertices do not yield a single-edge stitch and can be filtered out. Specifically, we utilize three observations here. First, register vertices can be ignored since memory error exploit cannot corrupt them. Second, the vertex must be defined (written) before the memory error and used (read) after the memory error. In Figure 4.3 (a), the code reads vertex (&uid, 3) before the memory error and writes vertices (&arg, 9) and the following one after the memory error. Therefore these three vertices are useless for single-edge stitches. Third, in the memory address dimension, the vertex address should belong to the memory region of the influence I. In our example, only vertex (a1, 4) falls into the intersection of the target flow and the influence area. We select this vertex for stitch.

StitchAlgo-1 shows the algorithm to identify single-edge stitch. From the given 2D-DFG, StitchAlgo-1 gets the target flow $TDFlow$ for the target vertex v_T. $TDFlow$ is in fact the largest connected subgraph of the whole 2D-DFG, containing the given target vertex v_T. For each vertex v inside $TDFlow$, the algorithm filters out the register vertex. Then it checks whether v is written before the memory error and read after the memory error. If so, it further checks whether v is under the influence of the memory error. If v passes all checks, we add the edge from memory error vertex to v into $\overline{E}$ as one possible solution. We consider the search space reduction due to our algorithm over a brute-force search for stitch edges. The naïve brute-force search would consider the Cartesian product of all vertices in the source flow and the target flow. In our algorithm, this search is reduced to the Cartesian product of only the live variables in the source flow at the time of corruption, and the vertices in the target flow as well as in I. In our experiments, we show that this reduction can be significant (see Section 4.5.2).

4.2.2 Advanced Stitching Technique

Single-edge stitch is a basic stitching method, creating one new edge. Advanced data-flow stitching techniques create paths with multiple edges in the new edge set $\overline{E}$. A *multi-edge stitch* can be synthesized through several ways. Attackers can use several single-edge stitches to create a multi-edge stitch. Another way is to perform *pointer stitch*, which corrupts a variable that is later used as a pointer to vertices in the source

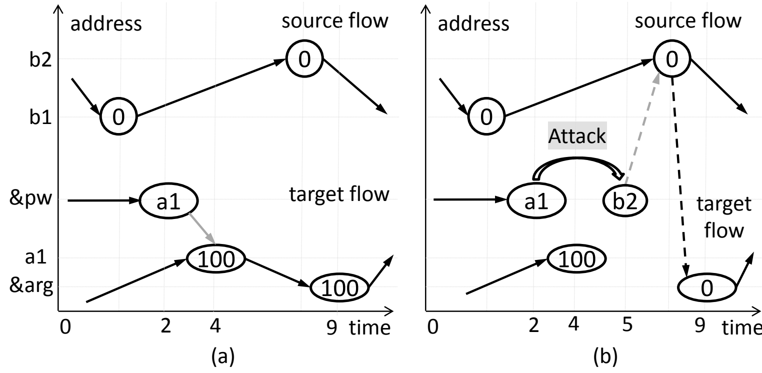

Figure 4.4: Two-edge stitch of `wu-ftpd`. The target flow is `pw->pw_uid`'s flow, and the source flow is the flow of a constant 0. The attack changes the variable `pw` at `&pw` from $a1$ to $b2$. A later operation reads 0 from $b2$ and pushes it on stack for `setuid`. Two edges are changed: one for pointer dereference and another for data movement.

or target flow. Since the pointer determines the address of the stitch source or the stitch target, corrupting the pointer introduces two different edges: one edge for the new "points to" relationship and one edge for the changed data flow. We revisit the example of `wu-ftpd` shown earlier in Code 4.2, illustrating a multi-edge stitch exploit in it. Instead of modifying the field `pw_uid`, we change its base pointer `pw` to an address of a structure with a constant `0` at the offset corresponding to the `pw_uid`. The vulnerable code then reads `0` and uses it as the argument of `setuid`, creating a privilege escalation attack. Figure 4.4 shows the 2D-DFGs for the benign and attack executions. Changing the value of `pw` creates two new edges (dashed lines): the grey edge that connects the corrupted pointer to a new variable it points to, and the black edge that writes the new variable into `setuid` argument. The result is a two-edge stitch.

Identifying Pointer Stitches. Our algorithm for finding multi-edge exploits using pointer stitching is shown in the StitchAlgo-2. The basic idea is to check each memory vertex in the source flow and the target flow. If it is pointed to by another vertex in the 2D-DFG, we select the pointer vertex to corrupt. The search for stitchable pointers on the target flow is different from that on the source flow. Specifically, for a vertex v in the target flow, we need to find an data edge (v', v) and a pointer vertex vp of v', and then change vp to point to a vertex vs in the source flow, so that a new edge (vs, v) will be created to stitch the data flows. For a vertex v in the source flow, we need to find an data edge (v, v') and a pointer vertex vp of v', and change vp to point to a vertex vt in the target flow, so that a new edge (v, vt) will be created to stitch the data flows. At

StitchAlgo-2: Pointer Stitch

```
Input: G: benign 2D-DFG,  I: memory influence,
       v_S: source vertex,   v_T: target vertex,
       cp: memory error vertex
Output: Ē: stitch edge candidate set
1  Ē = ∅
2  SrcFlow = subgraph(G, v_S)                 /* both data and address edges. */
3  TgtFlow = subgraph(G, v_T)
4  SDFlow = dataSubgraph(G, v_S)              /* only data edges */
5  TDFlow = dataSubgraph(G, v_T)
6  foreach v ∈ V(TDFlow) do
7      if isRegister(v) then continue
8      if ∄ (vi ∈ E(I)∧ (v, v') ∈ TDFlow) : vi.time < v'.time then continue
9      foreach (vp, v) ∈ E(TgtFlow) – E(TDFlow) do
           /* Only consider address edges.                                   */
10         if vp is used to write v then continue
           /* Expect data flow from v                                       */
11         foreach vs ∈ V(SDFlow) do
12             if ¬isRegister(vs) ∧ vs.isAliveAt(vp.time) then
13                 StitchAlgo-1(G, I, vp, cp, vs.address)
14 foreach v ∈ V(SDFlow) do
15     if isRegister(v) then continue
16     if ∀ vi ∈ I: v.time < vi.time then continue
17     foreach (vp, v) ∈ E(SrcFlow) – E(SDFlow) do
18         if vp is used to read v then continue
           /* Expect data flow into v                                       */
19         foreach vt ∈ V(TDFlow) do
20             if ¬ isRegister(vt) ∧∃(vt, v') ∈ TDFlow : vt.time < vp.time < v'.time then
21                 StitchAlgo-1(G, I, vp, cp, vt.address)
/* subgraph(G, v): the largest connected subgraph of G, containing v.      */
/* Here "connected" means connected by either data edges or address edges. */
/* v.isAliveAt(t): true if v still holds the same value at time t, false otherwise. */
/* Refer StitchAlgo-1 for meanings of other functions.                     */
```

the same time, we need to consider the liveness of the stitching vertices. For example, the source vertex should carry valid source data when it is used to write data out to the target vertex. Once we select the pointer vertex vp and its value (vt's or vs's address), the last step is to set the value into vp through the memory error exploit. StitchAlgo-2 invokes the basic stitching technique in StitchAlgo-1 to complete the last step.

Our technique uses vertex liveness and the memory error influence I to significantly reduce the search space. A naïve solution to finding pointer stitches would consider all pairs (vs, vt) where vs is in the source flow and vt is in the target flow. The search space will be the Cartesian product of the vertex set in the source flow (denoted as $V(SrcFlow)$) and the vertex set in the target flow (denoted as $V(TgtFlow)$). In contrast, in StitchAlgo-2, if the memory corruption occurs at time $t1$, the vertex used in the stitch edge from the source flow must be live at $t1$. Similarly, the vertex used in the

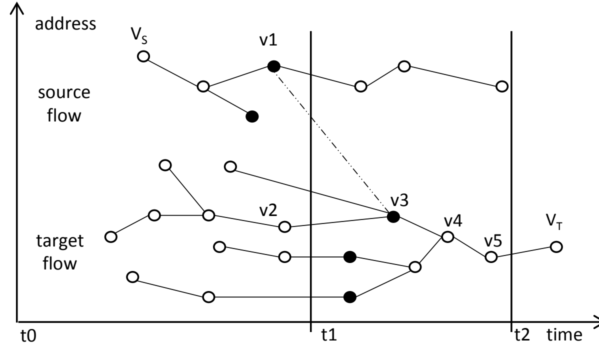

Figure 4.5: Stitch edge selection. The execution starts at time $t0$, and reaches memory error instructions at time $t1$. Target data is used at time $t2$, just before target vertex V_T. There are two stitch source candidates (black points in the source flow) and three stitch destination candidate (black points in the target flow). One of the stitch edge candidates is shown using the dotted line.

stitch edge from the target flow should be created after $t1$. We illustrate it in Figure 4.5, where only the black vertices are candidates. Furthermore, we restrict our search to the set of vertices whose pointer vertices vp are inside the memory influence as well. We call the selected vertices from the source flow *R-set*. Similarly, we call the vertices selected from the target flow *W-set*. Our algorithm reduces the search space to the Cartesian product of the R-set and W-set instead.

$$R\text{-}set = V(SrcFlow) \cap I, \quad W\text{-}set = V(TgtFlow) \cap I$$

$$|SS_{naive}| \;=\; |V(SrcFlow)| \times |V(TgtFlow)|$$

$$|SS_{pointer-stitch}| \;=\; |\; R\text{-}set \;| \times |\; W\text{-}set \;|$$

Pointer stitch constitutes a natural hierarchy of exploits, which can consist of multiple levels of dereferences of attacker-controlled pointers. For instance, in a *two-level pointer stitch* we can construct an exploit that corrupts a pointer vp_2 that points to the pointer vp. This can be achieved by treating vp as the target vertex, another pointer vp' holding the intended value (vt's or vs's address) as the source vertex and applying StitchAlgo-2 to change vp. In this case, StitchAlgo-2 is recursively used twice. Similarly, *N-level stitch* corrupts a pointer vp_N of the pointer $vp_{(N-1)}$ to make an attack (and so on), by applying StitchAlgo-2 N times recursively. Note that for a N-level stitch to work, we need to make sure the source vertex vp'_N "aligns" with the target vertex vp_N at each level, such that the program dereferences vp_N N-1 times to get the vertex vp,

and dereferences vp'_N N-1 time to get the intended value in the exploit.

Pointer stitch is one specific way to implement multi-edge stitches. In principle, it can be composed to create more powerful exploits, combining several other single-edge stitches in a "multi-step" stitch attack. In a multi-step stitch, several intermediate data flows are used to achieve data-flow stitching. Each step can be realized by pointer stitch or single-edge stitch. Multi-step stitch is useful when direct stitches of the source flow and the target flow are not feasible.

4.2.3 Challenges from ASLR

Address space layout randomization (ASLR) deployed by modern systems poses a strong challenge in mounting successful data-oriented attacks since vertex addresses are highly unpredictable. We develop two methods in data-oriented attacks to address this challenge: stitching with deterministic addresses and stitching by address reuse. Note that attackers can use others methods developed for control flow attacks to bypass ASLR here, like disclosure of random addresses [25, 148].

4.2.3.1 Stitching With Deterministic Addresses

When security-critical data is stored in deterministic memory addresses, stitching data flows of such data is not affected by ASLR. Existing work [5, 135, 162] have shown that current ASLR implementations leave a large portion of program data in the deterministic memory region. For example, Linux binaries are often compiled without the "*-pie*" option, resulting in deterministic memory regions. We study deterministic memory size of Ubuntu 12.04 (x86) binaries under directories `/bin`, `/sbin`, `/usr/bin` and `/usr/sbin`, and show the results in Table 4.1. Among 1093 analyzed programs, more than 87.74% have deterministic memory regions. Two hundred and twenty-three programs have deterministic memory regions larger than 64KB. Inside such memory regions, there is many security-critical data, like randomized addresses in `.got.plt` and configuration structures in `.bss`. Hence we believe stitch with deterministic addresses in real-world programs is practical.

We build an information leakage attack against the `orzhttpd` web server [8] (details in Section 4.5.4) using the stitch with deterministic addresses. To respond to a page request, `orzhttpd` uses a pointer to retrieve the HTTP protocol version string.

Table 4.1: Deterministic memory region size of binaries on Ubuntu 12.04 x86 system. Position-independent executables have size 0. Two largest numbers are highlighted for each directory.

size (KB)	/bin	/sbin	/usr/bin	/usr/sbin	Total
0	21	22	73	18	134
1 - 8	10	**33**	**150**	**20**	**213**
8 - 16	12	17	113	11	153
16 - 32	**23**	17	**147**	14	**201**
32 - 64	**19**	**22**	103	**25**	169
64 - 128	15	8	66	8	97
128 - 256	7	2	35	4	48
256 - 512	3	2	32	3	40
> 512	2	2	32	2	38
Total	112	125	751	105	1093

The pointer is stored in memory. If we replace the pointer value with the address of a secret data, the server will send that secret to the client. However this requires both the address of the pointer and the address of the secret to be predictable. In the `orzhttpd` example, we find that the address of the pointer is fixed (`0x8051164`) and choose the contents of the `.got.plt` section (allocated at a fixed address) as the secret to leak out. Figure 4.6 shows two 2D-DFGs for the benign execution and the attack, respectively. With this attack, the content of `.got.plt` is sent to the attacker, which leads to an memory address disclosure exploit useful for constructing second-stage control-hijacking attacks or stealing secret data in randomized memory region. Unlike a direct memory disclosure attack, here we use the corruption of deterministically-allocated data to leak randomized addresses.

Identifying Stitch with Deterministic Addresses. We represent the deterministic memory region as a set D. Our algorithm considers the intersection of D for the vertices in the source flow and the target flow. The previously outlined stitching algorithms can then be used directly prioritizing the vertices in the intersection with D.

4.2.3.2 Stitching By Address Reuse

If the security-critical data only exists inside the randomized memory region, data-oriented attacks cannot use deterministic addresses. To bypass ASLR in such cases,

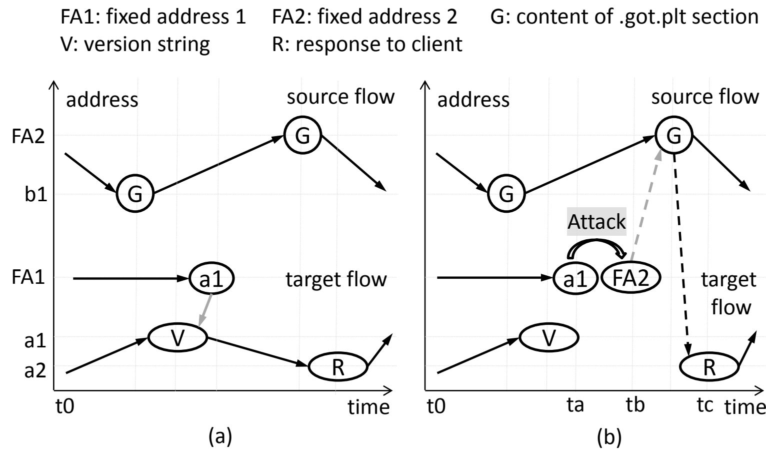

Figure 4.6: Stitch with deterministic memory addresses of the `orzhttpd` server. This attack is the similar to the one in Figure 4.4, except the address of the source vertex and the pointer's address of the target vertex are fixed. This attack works with ASLR.

we leverage the observation that a lot of randomized addresses are stored in memory. If we can reuse such real-time randomized addresses instead of providing concrete address in the exploit, the generated data-oriented attacks will be stable (agnostic to address randomization). There are two types of address reuse: partial address reuse and complete address reuse.

Partial Address Reuse. A variable's relative address, with respect to the module base address or with respect to another variable in the same module, is usually fixed. Attackers can easily calculate such relative addresses in advance. On the other hand, instructions commonly get a memory address with one base address and one relative offset (e.g., array access, switch table). If attackers control the offset variable, they can corrupt the offset with the pre-computed relative address from the selected vertex (source vertex or target vertex) and reuse the randomized base address. In this way attackers can access the intended data without knowing their randomized addresses. We show an example of a vulnerable instruction pattern, that allows the attacker partial ability to read a value from memory and write it out without knowing randomized addresses. If attackers control `%eax`, they can reuse the source base address `%esi` in the first instruction, and reuse the destination base address `%edi` in the second instruction. Any memory access instruction with a corrupted offset can be used to mount such reuse attack.

```
1  //attackers control %eax
2  mov   (%esi,%eax,4), %ebx   //reuse %esi
3  mov   %ecx, (%edi,%eax,4)   //reuse %edi
```

Complete Address Reuse. We observe that a variable's address is frequently saved in memory due to the limitation of CPU registers. If the memory error allows retrieving such spilled memory address for reading or writing, attackers can reuse the randomized vertex address existing in memory to bypass ASLR. For example, in the following assembly code, if attacker controls `%eax` on line 1, it can load a randomized address into `%ebx` from memory. Then, attacker can access the target vertex pointed by `%ebx` without knowing the concrete randomized address. The attacker merely needs to know the right offset value to use in `%eax` on line 2, or may have a deterministic `%esi` value to gain arbitrary control over addresses loaded on line 2.

```
1  //attacker controls %eax
2  mov  (%esi, %eax, 4), %ebx
3  mov  %ecx, (%ebx) / mov (%ebx), %ecx
```

```
1  struct user_details { uid_t uid; ... } ud;
2  ...                  //run with root uid
3  ud.uid = getuid();   //in get_user_info()
4  ...
5  vfprintf(...);       //in sudo_debug()
6  ...
7  setuid(ud.uid);      //in sudo_askpass()
8  ...
```

Code 4.3: Code snippet of `sudo`, setting uid to normal user id.

Let us consider a real example of the `sudo` program [77] that shows how to use such instruction patterns that permit complete address reuse meaningfully. Code 4.3 shows the related code of `sudo`, where a format string vulnerability exists in the `sudo_debug` function (line 5). At the time of executing `vfprintf()` on line 5, the address of the user identity variable (`ud.uid`) exists on the stack. The `vfprintf()` function with format string "%X\$n" uses the Xth argument on stack for "%n". By specifying the value of X, `vfprintf()` can retrieve the address of `ud.uid` from its ancestor's stack frame and change the `ud.uid` to the root user ID without knowing the stack base address. Figure 4.7 shows the 2D-DFGs for the benign execution and the attack. This attack works even if the fine-grained ASLR is deployed.

Identifying Stitch by Address Reuse. Memory error instructions for address reuse stitch should match the patterns we discuss above. For partial address reuse, the memory error exploit corrupts variable offsets, while for complete address reuse, the memory

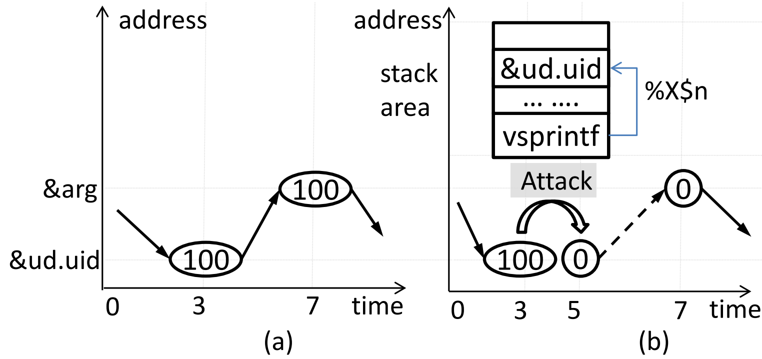

Figure 4.7: Stitch by complete memory address reuse of `sudo`. The dashed line is the new edge (single-edge stitch). An address of $ud.uid$ exists on ancestor's stack frame, which is reused to overwrite $ud.uid$.

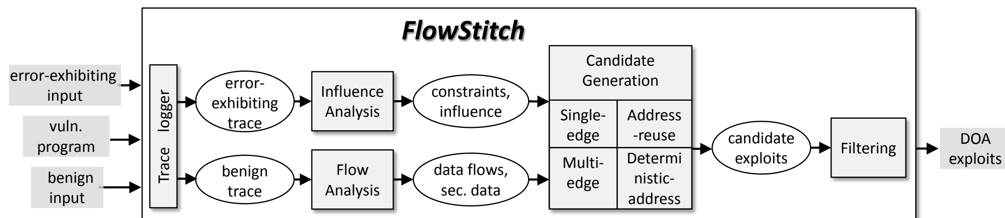

Figure 4.8: Overview of FLOWSTITCH. FLOWSTITCH takes a vulnerable program, an error-exhibiting input and a benign input of the program as inputs. It builds data-oriented attacks against the given program using data-flow stitching. Finally it outputs the data-oriented attack exploits.

error exploit can retrieve addresses from memory. Our approach intersects the memory error influence I with the source flow and the target flow. Then we search from the new source flow and the new target flow to identify matched instructions, from which we can build stitch by address reuse with methods discuss above.

4.3 The FLOWSTITCH System

We design a system called FLOWSTITCH to systematically generate data-oriented attacks using data-flow stitching. As shown in Figure 4.8, FLOWSTITCH takes three inputs: a program with memory errors, an error-exhibiting input, and a benign input of the program. The two inputs should drive the program execution down the same execution path until the memory error instruction, with the error-exhibiting input causing a crash. FLOWSTITCH builds data-oriented attacks using the memory errors in five steps. First, it generates the execution trace for the given program. We call the execution trace with the benign input the *benign trace*, and the execution trace with the error-exhibiting input the *error-exhibiting trace*. Second, FLOWSTITCH identifies the influence of the

memory errors from the error-exhibiting trace and generates constraints on the program input to reach memory errors. Third, FLOWSTITCH performs data-flow analysis and security-sensitive data identification using the benign trace. Fourth, FLOWSTITCH selects stitch candidates from the identified security-sensitive data flows with the methods discussed in Section 4.2. Finally, FLOWSTITCH checks the feasibility of creating new edges with the memory errors and validates the exploit. It finally outputs the input to mount a data-oriented attack.

FLOWSTITCH requires that the error-exhibiting input and the benign input follow the same code path until memory error happens. The reason is that FLOWSTITCH aligns the error-exhibiting trace and the benign trace to map the memory error instruction into the benign trace (FLOWSTITCH stitches data flows in the benign trace). Such pairs of inputs can be found by existing symbolic execution tools, like S2E [59], BAP [39] and SAGE [94], and fuzzing tools, like AFL [176], BFF [103], SymFuzz [51]. These tools explore multiple execution paths with various inputs. Before detecting one error-exhibiting execution, they usually have explored many matched benign executions. Previous work [98] discussed the method to select compatible benign input with the error one from a pool of benign inputs, which can be used here to help generate compatible error trace and benign trace. We plan to integrate FLOWSTITCH with memory error detection tools to support exploit generation on new memory errors.

4.3.1 Memory Error Influence Analysis

FLOWSTITCH analyzes the error-exhibiting trace to understand the influence I of the memory errors. It identifies two aspects of the influence: the time when the memory errors happens during the execution (temporal influence) and the memory range that can be written to in the memory error (spatial influence). From the error-exhibiting trace, FLOWSTITCH detects instructions whose memory dereference addresses are derived from the error-exhibiting input. We call these instructions *memory error instructions*. Note that data flows ending before such instructions or starting after them cannot be affected by the memory error, therefore they are out of the temporal influence.

Attackers get access to unintended memory locations with memory error instructions. However, the program's logic limits the total memory range accessible to attackers. To identify the spatial influence of the memory error instruction, we employ

dynamic symbolic execution techniques. We generate a symbolic formula from the error-exhibiting trace in which all the inputs are symbolic variables and all the path constraints are asserted true. Inputs that satisfy the formula imply that the execution to memory error instructions with an unintended address[3]. The set of addresses that satisfy these constraints and can be dereferenced at the memory error instruction constitute the spatial influence.

Note that FLOWSTITCH detects memory error instructions in the error trace, but connects data flows in the benign trace. Therefore we need to map the memory error instructions into the benign trace. To achieve this, we align the benign trace and the error trace. The alignment algorithm first generates the dynamic control flow graph for both trace, then it tries to match the functions starting from the entry point. It first matches the out-most function regardless of the internal instruction sequence of the callee function. Then it proceeds to match inner functions. With this method, the algorithm can tolerate minor execution difference in the callee functions. For example, `memcpy` function could have different number of iterations over the copy loop. Our method can successfully match callers of memcpy regardless of the length of the copy. Previous work [98, 139] also discussed similar trace alignment techniques.

4.3.2 Security-Sensitive Data Identification

As we discuss in Section 4.1.3, FLOWSTITCH synthesizes flows of security-sensitive data. There are four types of data that are interesting for stitching: input data, output data, program secret and permission flags. To identify input data, FLOWSTITCH performs taint analysis at the time of trace generation, treating the given input as an external taint source. For output data, FLOWSTITCH identifies a set of program sinks that send out the program data, like `send()` and `printf()`. The parameters used in sinks are the output data. Further, we classify program secret and permission flags into two categories: the program-specific data and the generic data. FLOWSTITCH accepts user specification to find out program-specific data. For example, user can provide addresses of security flags. For the generic data, FLOWSTITCH uses the following methods to automatically infer it.

- **System call parameters.** FLOWSTITCH identifies all system calls from the trace,

[3]This is true if the symbolic formula constructed is complete [95].

like `setuid`, `unlink`. Based on the system call convention, FLOWSTITCH collects the system call parameters.

- **Configuration data.** To identify configuration data, FLOWSTITCH treats the configuration file as a taint source and uses taint analysis to track the usage of the configuration data.

- **Randomized data.** FLOWSTITCH identifies stack canary based on the instructions that set and check the canary (e.g., storing canary in %gs:0x14 on Linux), and identifies randomized addresses if they are not inside the deterministic memory region (see below).

Deterministic Memory Region Identification. FLOWSTITCH identifies the deterministic memory region for stitch with deterministic addresses (Section 4.2.3.1). It first checks the program binary to identify the memory regions that will not be randomized at runtime. If the program is not position-independent, all the data sections shown in the binary headers will be at deterministic addresses. FLOWSTITCH collects loadable sections and gets a deterministic memory set D. This method works on both Linux and Windows system. FLOWSTITCH further scans benign traces to find all the memory writing instructions that write data into the deterministic memory set to identify data stored in such region. If the value written to such region is likely to be an address, the tool further checks whether the address falls into the deterministic region. The address outside such region is identified as randomized address. One of our goal is to leak such address value to bypass randomized for next attack.

Note that based on the functionality of the security-sensitive data, we predefine goals of the attacks. For example, the attack of `setuid` parameter is to change it to the root user's id 0. For a web server's home directory string, the goal is to set it to system root directory.

4.3.3 Stitching Candidate Selection

For identified security-sensitive data, FLOWSTITCH generates its data flow from the 2D-DFG. FLOWSTITCH selects the source flow originated from the source vertex V_S and the target flow ended at the target vertex V_T. It then uses the stitching methods dis-

cussed in Section 4.2 to find stitching solutions. Although any combination of stitching methods can be used here, FLOWSTITCH uses the following policy in order to produce a successful stitching efficiently.

1. FLOWSTITCH uses single-edge stitch technique before the multi-edge stitch technique. After the single-edge stitch's search space is exhausted, it seeks multi-edge stitch. FLOWSTITCH stops searching at four-edge stitch in our experiments.
2. FLOWSTITCH considers stitch with deterministic addresses before stitch by address reuse. After exhausting the search space of deterministic address and address reuse space, FLOWSTITCH continues searching stitches with concrete addresses shown in benign traces, for cases without ASLR.

4.3.4 Candidate Filtering

To overcome challenge **C3**, FLOWSTITCH checks the feasibility of each selected stitch edge candidate. We define the *stitchability constraint* to cover the following constraints.

- Path conditions to reach memory error instructions;
- Path conditions to continue to the target flow;
- Integrity of the control data;

The first two constraints are control-flow constraints and data-flow constraints, as we discussed in Section 3.2. Satisfying them guarantees that the execution path will reach both the memory error and the security-critical code. The last constraint guarantees that the execution will not violate the control flow integrity. To achieve the last constraint, we insert `assert` into the formula to make sure the control data are preserved. FLOWSTITCH generates the stitchability constraint using symbolic execution tools. The constraint is sent to SMT solvers as an input. If the solver cannot find any input satisfying the constraint, FLOWSTITCH picks the next candidate stitch edge. If it exists, the input will be the witness input that is used to exercise the execution path in order to exhibit the data-oriented attacks. Due to the concretization in symbolic constraint generation in the implementation, the constraints might not be complete [95], i.e., it may allow inputs that results in different paths. FLOWSTITCH concretely verifies the input generated by the SMT solver to check if it successfully mounts the data-oriented attacks on the program.

4.4 Implementation

We prototype FLOWSTITCH on Ubuntu 12.04 32 bit system. Note that as the first step the trace generation tool can work on both Windows and Linux systems to generate traces. Although the following analysis steps are performed on Ubuntu, FLOWSTITCH works for both Windows and Linux binaries.

Trace Generation. Our trace generation is based on the Pintraces tool provided by BAP [39]. Pintraces is a Pin [111] tool that uses dynamic binary instrumentation to record the program execution status. It logs all the instructions executed by the program into the trace file, together with the operand information. In our evaluation, the traces also contain dynamic taint information to facilitate the extraction of data flows.

Data Flow Generation. For input data and configuration data, FLOWSTITCH uses the taint information to get the data flows. To generate the data flow of the security-sensitive data, FLOWSTITCH performs backward and forward slicing on the benign trace to locate all the related instructions. It is possible for one instruction to have multiple source operands. For example, in `add %eax, %ebx`, the destination operand `%ebx` is derived from `%eax` and `%ebx`. In this case, one vertex has multiple parent vertices. As a result, the generated data flow is a graph where each node may have multiple parents.

Constraint Generation and Solving. The generation of the stitchability constraint required in Section 4.3.4 is implemented in three parts: path constraints, influence constraints, and CFI constraints. The stitchability constraint is expressed as a logical conjunction of these three parts. We use BAP to generate formulas which capture the path conditions and influence constraints. For control flow integrity constraint, we implement a procedure to search the trace for all the indirect `jmp` or `ret` instruction. Memory locations holding the return addresses or indirect jump targets are recorded. The control flow integrity requires that at runtime, the memory location containing control data should not be corrupted by the memory errors. The stitchability constraint is checked for satisfiability using the Z3 SMT-solver [81], which produces a witness input when the constraint is satisfiable.

4.5 Evaluation

Table 4.2: Experiment environments and benchmarks. # of Attacks gives the number of attacks generated by FLOWSTITCH, including privilege escalation attacks and information leakage attacks. FLOWSTITCH generates 19 data-oriented attacks from 8 vulnerable programs.

ID	Vul. Program	Vulnerability	Environment (32b)	# of Attacks	
				Priv.	Leak
CVE-2013-2028 [74]	nginx	stack buffer overflow	Ubuntu 12.04	1	1
CVE-2012-0809 [77]	sudo	format string	Ubuntu 12.04	1	0
CVE-2009-4769 [71]	httpdx	format string	Windows XP SP3	4	1
bugtraq ID: 41956 [146]	orzhttpd	format string	Ubuntu 9.10	1	1
CVE-2002-1496 [70]	null httpd	heap overflow	Ubuntu 9.10	2	0
CVE-2001-0820 [68]	ghttpd	stack buffer overflow	Ubuntu 12.04	1	0
CVE-2001-0144 [72]	SSHD	integer overflow	Ubuntu 9.10	2	1
CVE-2000-0573 [79]	wu-ftpd	format string	Ubuntu 9.10	2	1
Total	8 programs			14	5

In this section, we evaluate the effectiveness of data-flow stitching using FLOWSTITCH, including single-edge stitch, multi-edge stitch, stitch with deterministic addresses and stitch by address reuse. We also measure the search space reduction using FLOWSTITCH and the performance of FLOWSTITCH.

4.5.1 Efficacy in Exploit Generation

Table 4.2 shows the programs used in our evaluation, as well as their running environments and vulnerabilities. The trace generation phase is performed on different systems according to the tested program[4]. All generated traces are analyzed by FLOWSTITCH on a 32-bit Ubuntu 12.04 system. The vulnerabilities used for the experiments come from four different categories to ensure that FLOWSTITCH can handle different vulnerabilities. 7 of the 8 vulnerable programs are server programs, including HTTP and FTP servers, which are the common targets of remote attacks. The other one is the `sudo` program, which allows users to run command as another user on Unix-like system. The last 4 vulnerabilities were discussed in [56] to manually build data-oriented attacks. We apply FLOWSTITCH on these vulnerabilities to verify the efficacy of our method.

[4]On Windows XP, ASLR is not enabled. Our attack works if the system does not force to randomize the .bss section, which is common in current Windows systems.

Table 4.3: Evaluation of FlowStitch on generating data-oriented attacks. In the Attack Description column, L_i stands for information leakage attack, while M_i represents privilege escalation attack. Column 3 indicates whether the built attack can bypass ASLR or not. The "CP" column shows the number of memory error instructions. Trace size is the number of instructions inside the trace. The last 4 columns show the number of stitch sources and stitch targets before and after our selection. SrcFlow means source flow, while TgtFlow stands for target flow.

Vul. Apps	Attack Description	ASLR Bypass	CP	Error-exhibiting Trace Size	Benign Trace Size	# of nodes before		# of nodes after	
						SrcFlow	TgtFlow	SrcFlow	TgtFlow
nginx	L_0: private key		1	50789	411437	3	48	3	1
	M_0: http directory path				1717182	173	462	1	42
sudo	M_0: user id	✓	1	351988	854371	2083	1	1	1
httpdx	L_0: admin's password	✓	1	1197657	1361761	152	7	152	2
	M_0: admin's password	✓			1298247	78	120	1	8
	M_1: anon.'s permission	✓			1233522	78	2	1	1
	M_2: anon.'s root directory	✓			1522672	78	165	1	11
	M_3: CGI directory path	✓			1257694	78	480	1	30
orzhttpd	L_0: randomized address	✓	1	84694	131871	8	28	8	1
	M_0: directory path	✓			131871	368	95	1	19
null httpd	M_0: http directory path		2	160844	401285	3	141	2	47
	M_1: CGI directory path				335329	3	144	2	48
ghttpd	M_0: CGI directory path		1	312130	316473	3579	6	1	1
SSHD	L_0: root password hash		1	38201	3094592	776	56	97	2
	M_0: user id				674365	1	24	1	1
	M_1: authenticated flag				674365	1	2	1	1
wu-ftpd	L_0: env. variables		1	328108	1417908	88	5	88	1
	M_0: user id (single-edge)	✓			1057554	183	2	1	1
	M_1: user id (multi-edge)	✓			1057554	183	1	1	1

Results. Our result demonstrates that FlowStitch can effectively generate data-oriented attacks with different vulnerabilities on different platforms. The number of generated data-oriented attacks on each program is shown in Table 4.2 and their details are given in Table 4.3. FlowStitch generates a total of 19 data-oriented attacks for eight real-world vulnerable programs, more than two attacks per program on average. Among 19 data-oriented attacks, there are five information leakage attacks and 14 privilege escalation attacks. For the vulnerable `httpdx` server, FlowStitch generates five data-oriented attacks from a format string vulnerability.

Out of the 19 data-oriented attacks, 16 are previously unknown. The three known attacks are two `uid`-corruption attacks on `SSHD` and `wu-ftpd`, and a CGI directory corruption attack on `null httpd`, discussed in [56]. FlowStitch successfully re-

produces known attacks and builds new data-oriented attacks with the same vulnerabilities. Note that FLOWSTITCH produces a different `ghttpd` CGI directory corruption attack than the one described in [56]. Details of this attack are discussed in Section 4.5.4. The results show the efficacy of our systematic approach.

From our experiments, seven out of 19 of the data-oriented attacks are generated using multi-edge stitch. The significant number of new data-oriented attacks generated by multi-edge stitch highlights the importance of a systematic approach in managing the complexity and identifying new data-oriented attacks. As a measurement of the efficacy of ASLR on data-oriented attacks, we report that 10 of 19 attacks work even with ASLR deployed. Among 10 attacks, two attacks reuse randomized addresses on the stack and eight attacks corrupt data in the deterministic memory region. We observe that security-sensitive data such as configuration option is usually represented as a global variable in C programs and reside in the `.bss` segment. This highlights the limitation of current ASLR which randomizes the stack and heap addresses but not the `.bss` segment.

For three of 19 attacks, FLOWSTITCH requires the user to specify the security-sensitive data, including the private key of `nginx`, the root password hash and the authenticated flag of `SSHD`. For others, FLOWSTITCH automatically infers the security-sensitive data using techniques discussed in Section 4.3.2. Once such data is identified, FLOWSTITCH automatically generates data-oriented exploits.

4.5.2 Reduction in Search Space

Data-flow stitching has a large search space due to the large number of vertices in the flows to be stitched. Manual checking through a large search space is difficult. For example, in the root password hash leakage attack against `SSHD` server, there are 776 vertices in source flow containing the hashed root passwords. In the target flow, there are 56 vertices leading to the output data. Without considering the influence of the memory errors, there are a total of 43,456 possible stitch edges. After applying the methods described in Section 4.2, we get the intersection of the memory error influence I with the stitch source set R-set and the stitch target set W-set. In this way, the number of candidate edges is reduced from 43,456 to 194, obtaining a reduction ratio of 224.

The last four columns in Table 4.3 give the detailed information of the search space for each attack. For most of the data-oriented attacks, there is a significant reduction

Table 4.4: Performance of trace and flow generation using FLOWSTITCH. The unit used in the table is second, so 1:07 means one minute and seven seconds.

Attacks		nginx		sudo	httpdx					orzhttpd	
		L_0	M_0	M_0	L_0	M_0	M_1	M_2	M_3	L_0	M_0
TraceGen	error	0:08		0:35	0:08					0:17	
	benign	0:22	0:36	1:07	0:45	0:51	0:50	1:03	0:53	0:20	0:20
Slicing	error	0:06		1:17	0:12					0:12	
	benign	2:41	0:12	3:34	5:56	4:44	4:52	4:45	4:47	0:24	1:04
Total		3:17	1:02	6:33	7:01	5:55	6:02	6:08	6:00	1:13	1:53

Attacks		nullhttpd		ghttpd	SSHD			wu-ftpd			avg
		M_0	M_1	M_0	L_0	M_0	M_1	L_0	M_0	M_1	
TraceGen	error	0:13		0:09	2:35			0:12			0:32
	benign	1:20	0:52	0:18	9:38	5:30	5:30	0:50	0:31	0:31	1:41
Slicing	error	0:14		0:12	1:02			0:19			0:26
	benign	6:21	2:29	0:09	21:08	1:22	1:00	5:42	0:27	0:26	3:47
Total		8:08	3:48	0:48	34:23	10:29	10:07	7:03	1:29	1:28	6:27

in the number of possible stitches. `ghttpd`-M_0 achieves the highest reduction ratio of 21,474 while `SSHD`-M_1 achieves the lowest reduction ratio of two. The median reduction ratio is 183 achieved by `wu-ftpd`-M_1(multi-edge). Given the relatively large spatial influence of the memory error, most of the reduction is achieved by the temporal influence of I.

4.5.3 Performance

We measure the time FLOWSTITCH uses to generate data-oriented attacks. Table 4.4 shows the results, including the time of trace generation and the time of data-flow collection (slicing). Note that the trace generation time includes the time to execute instructions that are not logged (e.g., `crypto` routines and `mpz` library for `SSHD`). As we can see from Table 4.4, FLOWSTITCH takes an average of six minutes and 27 seconds to generate the trace and flows. Most of them are generated within 10 minutes. The information leakage attack of `SSHD` server takes the longest time, 34 minutes and 23 seconds, since crypto routines execute a large number of instructions. From the performance results, we can see that the generation of data flows through trace slicing takes up most of the generation time, from 20 percent to 87 percent. Currently, our slicer works on BAP IL file. We plan to optimize the slicer using parallel tools.

4.5.4 Case Studies

We present five case studies to demonstrate the effectiveness of stitching methods and interesting observations.

Sensitive Data Lifespan. A common defense employed to reduce the effectiveness of data-oriented attacks is to limit the lifespan of security-critical data [56, 60]. This case study highlights the difficulty of doing it correctly. In the implementation of `SSHD`, the program explicitly zeros out sensitive data, such as the RSA private keys, as soon as they are not in use. For password authentication on Linux, `getspnam()` provided by glibc is often used to obtain the password hash. Rather than using the password hash directly, `SSHD` makes a local copy of the password hash on stack for its use. Although the program makes no special effort is to clear the copy on the stack, the password on stack is eventually overwritten by subsequent function frames before it can be leaked. The developer explicitly deallocates the original hash value using `endspent()` [3] in the glibc internal data structures. However, glibc does not clear the deallocated memory after `endspent()` is called and this allows FLOWSTITCH to successfully leak the hash from the copy held by glibc. Hence, this case study highlights that sensitive information should not be kept by the program after usage, and that identifying all copies of sensitive data in memory is difficult at the source level.

Multi-edge Stitch – ghttpd CGI Directory. The `ghttpd` application is a lightweight web server supporting CGI. A stack buffer overflow vulnerability was reported in version 1.4.0 - 1.4.3, allowing remote attackers to smash the stack of the vulnerable `Log()` function. During the security-sensitive data identification, FLOWSTITCH detects `execv()` is used to run an executable file. One of `execv()`'s arguments is the address of the program path string. Controlling it allows attackers to run arbitrary commands. FLOWSTITCH is unable to find a new data dependency edge using single-edge stitching, since there is no security-sensitive data on the stack frame to corrupt. FLOWSTITCH then proceeds to search for a multi-edge stitch. For the program path parameter of `execv()`, FLOWSTITCH identifies its flow, which includes use of a series of stack frame-base pointers saved in memory. The temporal constraints of the memory error exploit only allow the saved `%ebp` of the `Log()` function to be corrupted. Once

the `Log()` function returns, the saved `%ebp` is used as a pointer, referring to all the local variables and parameters of `Log()` caller's stack frame. FLOWSTITCH corrupts the saved `%ebp` to change the variable for the CGI directory used in `execv()` system call. This attack is a four-edge stitch by composing two pointer stitches.

Chen *et al.* [56] discussed a data-oriented attack with the same vulnerability, which was in fact a two-edge stitch. However, that attack no longer works in our experiment. The `ghttpd` program compiled on our Ubuntu 12.04 platform does not store the address of command string on the stack frame of `Log()`. Only the four-edge stitching can be used to attack our `ghttpd` binary.

Bypassing ASLR – orzhttpd Attacks. The `orzhttpd` web server has a format string vulnerability which the attacker can exploit to control almost the whole memory space of the vulnerable program. FLOWSTITCH identifies the deterministic memory region and the randomized address on stack under `fprintf()` frame. The first attack which bypasses ASLR is a privilege escalation attack. This attack corrupts the web root directory with single-edge stitching and memory address reuse. The root directory string is stored on the heap, which is allocated at runtime. FLOWSTITCH identifies the address of the heap string from the stack and reuses it to directly change the string to `/` based on the pre-defined goal (Section 4.3.2). The second attack is an information leakage attack, which leaks randomized addresses in the `.got.plt` section. FLOWSTITCH identifies the deterministic memory region from the binary and performs a multi-edge stitch. The stitch involves modifying the pointer of an HTTP protocol string stored in a deterministic memory region. FLOWSTITCH changes the pointer value to the address of `.got.plt` section and a subsequent call to send the HTTP protocol string leaks the randomized addresses to attackers.

Privilege Escalation – Nginx Root Directory. The `Nginx` HTTP server 1.3.9-1.4.0 has a buffer overflow vulnerability [74]. FLOWSTITCH checks the local variables on the vulnerable stack and identifies two data pointers that can be used to perform arbitrary memory corruption. The memory influence of the overwriting is limited by the program logic. FLOWSTITCH identifies the web root directory string from the configuration data. It tries single-edge stitching to corrupt the root directory setting. The root

directory string is inside the memory influence of the arbitrary overwriting. FLOWSTITCH overwrites the value `0x002f` into the string location, thus changing the root directory into `/`. FLOWSTITCH verifies the attack by requesting `/etc/passwd` file. As a result, the server sends the file content back to the client.

Information Leakage – httpdx Password. The `httpdx` server has a format string vulnerability between version 1.4 to 1.5 [71]. The vulnerable `tolog()` function records FTP commands and HTTP requests into a server-side log file. Note that direct exploitation of this vulnerability does not leak information. Using the error-exhibiting trace, FLOWSTITCH identifies the memory error instruction and figures out that there is almost no limitation on the memory range affected by attackers. From the `httpdx` binary, FLOWSTITCH manages to find a total of 102MB of deterministic memory addresses. From the benign trace, FLOWSTITCH generates data flows of the root user passwords. This is the secret to be leaked out. The FLOWSTITCH generates the necessary data flow which reaches the `send()` system call automatically.

Starting from the memory error instruction, FLOWSTITCH searches backwards in the secret data flow and identifies vertices inside the deterministic memory region. FLOWSTITCH successfully finds two such memory locations containing the "admin" password: one is a buffer containing the whole configuration file, and another only contains the password. At the same time, FLOWSTITCH searches forwards in the output flow to find the vertices that affect the buffer argument of `send()`. Our tool identifies vertices within the deterministic memory region. The solver gives one possible input that will trigger the attack. FLOWSTITCH confirms this attack by providing the attack input to the server and receiving the "admin" user password.

4.6 Related Work

Data-Oriented Attack. Several work [63, 129, 160, 166, 177, 180, 182] has been done to improve the practicality of CFI, increasing the barrier to constructing control flow hijacking attacks. Instead, data-oriented attacks are serious alternatives. Data-oriented attacks have been conceptually known for a decade. Chen *et al.* constructed data-oriented attacks to show that data-oriented attack is a realistic threat [56]. However, no systematic method to develop data-oriented attacks is known yet. In our work, we

develop a systematic way to search for possible data-oriented attacks. This method searches attacks within the candidate space efficiently and effectively.

Automatic Exploit Generation. Brumley *et al.* [41] described an automatic exploit generation technique based on program patches. The idea is to identify the difference between the patched and the unpatched binaries, and generate an input to trigger the difference. Avgerinos *et al.* [23] discussed Automatic Exploit Generation(AEG) to generate real exploits resulting in a working shell. Felmetsger *et al.* [88] discussed automatic exploit generation for web applications. The previous work focused on generating control flow hijacking exploits. FLOWSTITCH on the other hand generates data-oriented attacks that do not violate the control flow integrity. To our knowledge, FLOWSTITCH is the first tool to systematically generate data-oriented attacks.

Defenses against Data-Oriented Attacks. Data-oriented attacks can be prevented by enforcing data-flow integrity (DFI). Existing work enforces DFI through dynamic information tracking [84, 170, 173] or by legitimate memory modification instruction analysis [50, 178]. However, DFI defenses are not yet practical, requiring large overheads or manual declassification. An ultimate defense is to enforce the memory safety to prevent the attacks in their first steps. Cyclone [105] and CCured [124] introduce a safe type system to the type-unsafe C languages. SoftBound [122] with CETS [123] uses bound checking with fat-pointer to force a complete memory safety. Cling [17] enforces temporal memory safety through type-safe memory reuse. Data-oriented attack prevention requires a complete memory safety.

4.7 Summary

In this work, we present a new concept called data-flow stitching, and develop a novel solution to systematically construct data-oriented attacks. We discuss novel stitching methods, including single-edge stitch, multi-edge stitch, stitch with deterministic addresses and stitch by address reuse. We build a prototype of data-flow stitching, called FLOWSTITCH. FLOWSTITCH generates 19 data-oriented attacks from eight vulnerable programs. Sixteen attacks are previously unknown attacks. All attacks bypass DEP and the CFI checks, and 10 bypass ASLR. The result shows that automatic generation of

data-oriented exploits exhibiting significant damage is practical.

Chapter 5

Exploiting Memory Errors with Data-Oriented Programming

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 [33, 35, 37, 52, 149]. In response, numerous principled defenses for control-hijacking attacks have been proposed. Examples of these include control-flow integrity (CFI) [15, 130, 160, 163, 180, 182], protection of code pointers (CCFI, CPI) [109, 113], timely-randomization of code pointers (TASR) [32], memory randomization [134], and write-xor-execute (W⊕X, or data-execution prevention, DEP) [21]. 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 [32, 113] or by directly checking the targets of control transfers [15, 63, 129, 130, 160, 180, 182]. 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 *data-oriented attacks* [56], are known to cause significant damage — such as leakage of secret keys (HeartBleed) [13], enabling untrusted code import in browsers [175], and privilege escalation in servers. However, data-oriented attacks provide limited expressiveness in attack payloads (e.g., allowing corruption or leakage of a few security-critical data bytes). We define *expressiveness* as the ability to perform various type of computations. The ultimate expressiveness is Turing-completeness, where any computation can be performed freely. Previous data-oriented exploits may achieve specific functionality, but fail to provide the capability for general computations.

In this work, we show that data-oriented attacks with rich expressiveness can be crafted using systematic construction techniques. We demonstrate that data-oriented 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 [149]. ROP attacks are known to be Turing-complete because of a similar systematic construction [102, 149]. 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 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 work:

- *DOP.* We propose data-oriented programming (DOP), a general method to build Turing-complete data-oriented 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 data-oriented attacks 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 data-oriented attacks 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/*.

5.1 Problem

5.1.1 Background: Data-Oriented Attacks

Data-oriented 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 [56]. My previous work provides a general construction method to automatically synthesize simple payloads to effect such attacks. The attack payloads either writes a target variable of choice or leaks contents of a sensitive memory region.

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 [10, 175]. 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 [47]. 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 data-oriented 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 [136] [1] and Windows [118] [2] sanitizes or blocks the use of %n, which severely limits the expressiveness of the attack. The question about how expressive are data-oriented attacks arising from common memory errors in arbitrary pieces of code is not well-understood. Since data-oriented attacks cannot divert the control flow to arbitrary locations, unlike ROP attacks [102, 149], the expressiveness is believed to be very limited.

5.1.2 Example of Data-oriented Programming

In fact, data-oriented attacks can offer rich exploits from common vulnerabilities. To see an example, consider the vulnerable code snippet shown in Code 5.1. 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 5.1: Vulnerable FTP server with data-oriented gadgets.

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

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

```
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 5.2: 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 5.1.

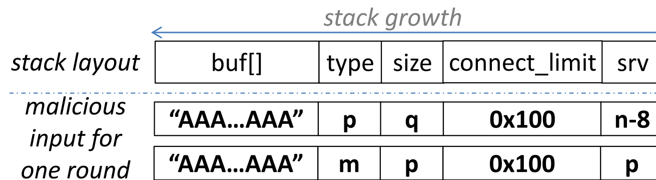

Figure 5.1: Malicious input to trigger the loop body in Code 5.2 by stitching data-oriented gadgets in Code 5.1. The upper side is the stack layout of Code 5.1. Refer Table 5.1 for details of p, q, m and n.

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 Code 5.1 would allow the remote adversary to simulate the function shown in Code 5.2, which maintains a linked list of integers in memory and increments each integer by a desired value. Table 5.1 illustrates how the code in the loop body gets simulated with the malicious input in Figure 5.1. Attackers can repeatedly send the same input sequence to implement the `updateList` function in Code 5.2.

This data-oriented 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

Table 5.1: Simulating the loop body in Code 5.2 with the data-oriented gadgets in Code 5.1. In column "Simulated Instr.", highlighted instructions are useful for the simulation, while other instructions are side effects of the attack.

Overflow	Executed Instr. (Code 5.1)	Simulated Instr. (Code 5.2)
type ← p	if(*type == NONE) break;	**if(list == NULL) break;**
size ← q	srv->typ = *type;	**srv = list;**
srv ← n-8	srv->total += *size;	**list->prop += addend;**
type ← m	if(*type == NONE) break;	if(list == NULL) break;
size ← p	if(*type == STREAM)	if(list == STREAM)
srv ← p	*size = *(srv->cur_max);	**list = list->next;**
p – &list; q – &addend; m – &STREAM; n – &srv		

security-critical data or functions to enable such attack. The control flow conforms to the precise CFG.

5.1.3 Research Questions

In this work, we aim to answer the following questions about data-oriented attacks:

- **P1**: How often do data-oriented gadgets arise in real-world programs? How often do gadget dispatchers exist?
- **P2**: Is it possible to chain gadgets for a desired computation? Can attackers build Turing-complete attacks with this method?
- **P3**: What is the security implication of this attack method for current defense mechanisms?

5.2 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.

5.2.1 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

Table 5.2: MINDOP language. To simulate (conditional) jump, data-oriented gadget changes the virtual input pointer (vpc) accordingly.

Semantics	**Expressions 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		

flow with respect to the program CFG. As shown in Section 5.1.2, the key is to manipulate non-control data such that the executed 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 5.4 confirms. MINDOP is Turing-complete, which we establish in Section 5.2.4.

The MINDOP language (shown in Table 5.2) 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.

5.2.2 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).

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.

Data-oriented gadgets are similar to code gadgets employed in return-oriented programming (ROP) [149], or in jump-oriented programming (JOP) [35]. 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

Table 5.3: 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**

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 5.3 shows one example of addition gadget with C and the assembly representation. This gadget is modeled from the real-world program ProFTPD [9]. 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 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

Table 5.4: 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 5.5: 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`

an assignment gadget is shown in Table 5.4.

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 5.2, 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 5.5 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.

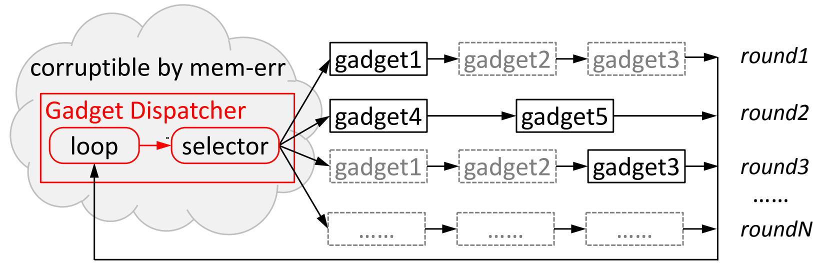

Figure 5.2: The design model of DOP in MINDOP. The gadget dispatcher includes a loop and a selector. The loop keeps the program passing by the selector and various data-oriented gadgets. For each round, the selector is controlled by the memory error to activate particular data-oriented gadgets.

5.2.3 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 5.1, line 6 and 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 5.2 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 5.1 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 5.3 taken from the real ProFTPD program [9][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 sim-

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

```
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 5.3: Example data-oriented gadget of jump operation.

ulate 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.

The virtual PC in non-interactive mode requires a dedicated space for malicious input and a controllable input pointer. In Section 5.4 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.

5.2.4 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 $Q \times \Sigma \mapsto \Sigma \times \{L, R\} \times Q$

```
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 5.4: Data-oriented gadget sequence to initialize the Turing Machine.

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 Code 5.4. This requires three basic types of gadgets: assignment (`MOV`), dereference (`LOAD` and `STORE`) and addition (`ADD`).

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. Code 5.5 shows

Table 5.6: 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	-	-

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-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. One example of transition table for simulating a Turing machine that shifts the given input by one bit is shown in Table 5.6. 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.

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 5.5 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

```
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 5.5: Gadget sequence to simulate one step in the Turing Machine.

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.

5.3 DOP Attack Construction

Constructing DOP attacks against a program requires a concrete memory error and specification of the malicious behavior. Our analysis first identifies 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.

5.3.1 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. Further, a useful gadget has to satisfy particular requirements. We use static analysis as an aid in identifying these gadgets.

Algorithm 3: Data-oriented gadget identification.

```
Input: G:- the vulnerable program
Output: S:- data-oriented gadget set
1  S = ∅;
2  FuncSet = getFuncSet(G)          /* getFuncSet(G): the set of functions in the program. */
3  foreach f ∈ FuncSet do
4      cfg = getCFG(f)              /* getCFG(f): the control flow graph (CFG) of the function. */
5      for instr = getNextInstr(cfg) do      /* getNextInstr(cfg): next instruction in CFG. */
6          if isMemStore(instr) then   /* isMemStore(instr): true if the instr writes memory. */
7              gadget = getBackwardSlice(instr, f)     /* the backward slice of instr in f. */
8              input = getInput(gadget)          /* getInput(gadget): the input of the gadget. */
9              if isMemLoad(input) then          /* true if the input is loaded from memory */
10                 S = S ∪ {gadget}
/* The same meaning applies to Algorithm 4.                                                */
```

- **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 depends on concrete memory errors. We need to find malicious input that makes the program execute selected gadgets with the expected addresses and order. During exploit, we avoid corrupt control data in the memory. Since data-oriented programming corrupts substantial memory locations, we also need to avoid program crashes.

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

5.3.2 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 5.2.2.
- **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. Algorithm 3 shows our method for data-oriented gadget identification. Our analysis iterates through all functions in the program. For each function, we scan through the CFG to find store instructions (line 6). We treat each store instruction 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 (line 7). The generated data-flow contains the instructions that derive the operands. Note that the backward slice is performed within the function. Therefore the slice stops at the function boundary. Then we check all the inputs to the slice (line 8). If any input is loaded from memory (line 9), we mark it as a data-oriented gadget (line 10).

Gadget Classification. We classify data-oriented gadgets into different categories based on their semantics and computed 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 5.2.2). 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

Algorithm 4: Gadget dispatcher identification.

```
   Input: G:- the vulnerable program
   Output: D:- gadget dispatcher set
1  D = ∅;
2  FuncSet = getFuncSet(G)
3  foreach f ∈ FuncSet do
4      foreach loop = getLoop(f) do        /* getLoop(f): the set of loops inside function f. */
5          loop.gadgets = ∅
6          foreach instr = getNextInstr(loop) do
7              if isMemStore(instr) then
8                  loop.gadgets ∪= getGadget(instr)  /* retrieve the gadget associated with instr */
9              else if isCall(instr) then        /* isCall(instr): true if instr is call instruction */
10                 target = getTarget(instr)                                /* get the call target */
11                 loop.gadgets ∪= getGadget(target)
12         if loop.gadgets != ∅ then
13             D = D ∪ {loop}
   /* Refer Algorithm 3 for other functions.                                                    */
```

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.

5.3.3 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 4 gives the dispatcher identification algorithm. Similar to Algorithm 3, our search covers all functions. Since loops are necessary for attackers to repeatedly connect gadgets, we first identify all loops in the function (line 4). For each loop, we scan the instructions in the loop body to find interesting gadgets with Algorithm 3 (line 6 - 8). For function calls within the loop, we step into functions through the call graph and iterate through all instructions inside (line 9 - 11). This gives us an over-approximate set of gadget candidates for a particular dispatcher. If the loop contains at least one useful gadget (line 12), we add it into the dispatcher set (line 13). 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

[4] Like the cases in Section 5.4

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 dispatchers from static analysis and dynamic analysis as the final result.

5.3.4 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. Based on the gadget classification, we complete the stitching steps manually, with the following method. Due to the limitation of human effort, the manual construction may miss some possible attacks. However, our goal is to demonstrate the expressiveness of data-oriented attacks, instead of soundness. We leave the last step of the automatic construction as the future work.

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

5.4 Evaluation

In this section, we measure the feasibility of data-oriented programming and answer the research questions outlined in Section 5.1.3. We first show the prevalence of data-oriented gadgets and gadget dispatchers in real-world x86 programs (**P1**). 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 (**P2**). 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 (**P3**).

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 [79], ProFTPD [9]), HTTP server (nginx [74]), daemons (bitcoind [1], sshd [11]), network packet analyzer (Wireshark [14]), user library musl libc [5] [7], and common user utilities (mcrypt [6], sudo [12]).

5.4.1 Feasibility of DOP

We study our 9 applications and measure the number of x86 gadgets that 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.

[5]We analyze standard C library musl libc instead of glibc [4] because glibc cannot be compiled with LLVM. In our analysis, we use BusyBox [2] built against musl libc.

Table 5.7: **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**

- 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 5.7 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 than one dispatcher (See Column 4 in Table 5.7). 1443 of these dispatchers contain x86 gadgets of our interest (See Column 5 in Table 5.7). 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 (see Section 5.3.2). Attackers can control the inputs of global gadgets from a memory error at any location. For function-parameter gadgets, attackers can control their inputs only if the memory error occurs before the parent function. We currently ignore the local gadgets as it has a strict requirement. 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 5.7 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.

Table 5.8: **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-	Load /	Arith-	Logical	Condi-	Total	Virtual	Is
Name	**CVE**	**Type**	**Cap**	**Exec**	**Exec**	**Used**	**ment**	**Store**	**metic**		**tional**	**Gadgets**	**PC?**	**TC?**
bitcoind	2015-6031 [65]	SBO	SW	0	0	0	0	0	0	0	0	0		
musl libc + ping	2015-1817 [145]	SBO	AW	83	88	16	18 ✓	45	6 ✓	19 ✓	0	88		
Wireshark	2014-2299 [76]	SBO	AW	152	44	1	1 ✓	1 ✓	2 ✓	0	1 ✓	5	✓	✓
nginx	2013-2028 [74]	SBO	AW	82	91	30	39	441	68 ✓	119	11	678		
mcrypt	2012-4409 [75]	SBO	AW	31	10	2	1 ✓	0	5	1	0	7		
sudo	2012-0809 [77]	FSV	AW	27	9	2	3	12	4 ✓	2	2	23		
ProFTPD	2006-5815 [78]	SBO	AW	146	92	31	15 ✓	69 ✓	22 ✓	18	2 ✓	126	✓	✓
sshd	2001-0144 [72]	IO	AW	91	34	20	10	11	19 ✓	120	0	160		
WU-FTPD	2000-0573 [79]	FSV	AW	23	36	9	47 ✓	21 ✓	102 ✓	8	8	186	✓	
			TOTAL	635	404	110	134	600	228	287	24	**1273**	**3**	**3**

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 5.8). 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. The gadgets inside the invoked functions and enfolding loops are the reachable gadgets from the dispatcher. Table 5.8 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. A reachable gadget does not necessarily imply a corruptible 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 5.8). 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 5.8). 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 trig-

gerable 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 5.8. We have also constructed end-to-end attacks using some of these gadgets as discussed in Section 5.4.3.

5.4.2 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. Automatic construction of DOP attacks are possible, like our previous work. However, it is much challenging to achieve as DOP requires to stitch hundreds of small gadgets. This is not a problem here as our goal is to demonstrate the expressiveness of data-oriented programming, instead of providing an automatic construction method. 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 5.4.3 first, where we show expressive attacks with these Turing-complete gadgets.

5.4.2.1 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 [78]). Line 14 in Code 5.6 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.

- *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 5.6), the

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

```
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 5.7: 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.

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.

- *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 5.7), 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 5.8: 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 5.8 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.
- *(Conditional) jump operation*. Code 5.9 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 5.9) 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 5.4.3 we show the case studies of expressive exploits against ProFTPD.

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

5.4.2.2 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 [76]). The fixed-size buffer `pd` (shown on Line 5 of Code 5.10) 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.

- *Assignment operation*. We identify an assignment operation from the function `packet_list_change _record`, called after the memory error function. Line 16 in Code 5.10 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 5.10 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`

```
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 5.10: Wireshark code snippet of the vulnerable function and gadgets.

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 5.10 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 5.10. 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.

5.4.3 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.

5.4.3.1 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 [134]. 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 5.3. 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!

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

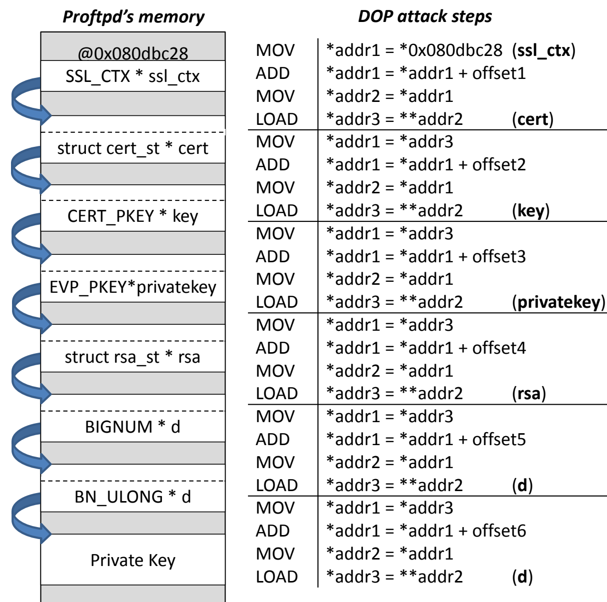

Figure 5.3: Pointer dereference chain and malicious MINDOP program in attack against ProFTPD. The attack requires 8 memory dereferences from the deterministic location to the private key. Each dereference is implemented by 4 gadgets.

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 [78] to simulate the malicious MINDOP program (as shown in Figure 5.3), 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`) [32]. 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.

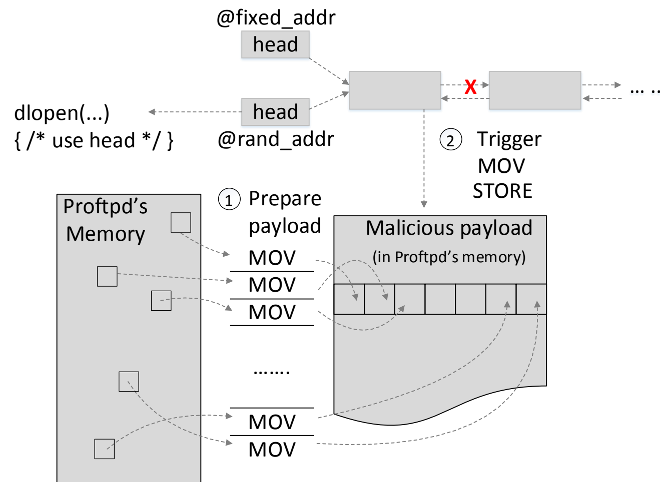

Figure 5.4: Simulating a network bot. There are two steps in this attack: ① Prepare the payload in Proftpd's memory; ② Trigger the memory error. Each step uses many data-oriented gadgets.

Secondly, non-interactive DOP attacks can perform all the necessary computation in-place between two system-calls. Even if the same techniques are deployed on data pointer, DOP can still bypass them. For example, given simple programs in Code 5.11 and Code 5.12 (in Section 5.5.2.1 and Section 5.5.2.2), TASR cannot defend it against DOP attacks. We refer interested readers to Section 5.5.2.3 for details.

5.4.3.2 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 MIN-DOP (also see Shapiro *et al.* [150]). 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 5.4). 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 5.4). 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.

5.4.3.3 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 [182] and DEP uses non-executable memory regions to prevent code injection attacks [21]. 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 [182]) or fixed-tag based solutions (assuming non-writable code region) [15], to effect control-hijacking exploits in ProFTPD. We refer interested readers to Section 5.5.2.3 for details.

5.4.4 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.

5.5 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.

5.5.1 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 [134]. 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 [32], 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 [62], 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 [15]. BinCFI relies on non-writable address translation table to enforce target checking [182]. 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 5.4.3.3. 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 [109, 113]. Code pointer integrity (CPI) is one such defense, which is based on memory isolation [109]. 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 [165]). 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 [113]. 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 work'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 Section 5.5.2.1).

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 5.4.4.

5.5.2 Example Vulnerable Programs

5.5.2.1 A Program Allowing DOP to Bypass TASR and CCFI

Code 5.11 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 [32, 113]. There are two data pointers, `pms1` and `pms2` and they are also pointing

```
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 5.11: 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.

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 data-oriented attacks to swap the value of `pms1` and `pms2`. Then the code in line 20 will call function `m2` instead.

5.5.2.2 Another Program Allowing DOP to Bypass TASR

Code 5.12 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.

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

5.5.2.3 Using DOP to Break CFI Implementations and DEP

BinCFI. BinCFI uses a read-only table to store all legitimate function entries and call-sites [182]. 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 5.13 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 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 [21] 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 [15]. Specifi-

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

cally, 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. The relocation functionality provided by DOP allows to temporally change the permission of any page to writable, including code pages and read-only data pages. Then it updates the page content based on arguments. Finally it changes the permission back. Attackers can abuse this functionality to 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.

5.5.3 Potential Defenses for DOP

5.5.3.1 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 [105, 124]. SoftBound with CETS stores metadata for each pointer inside a disjointed memory for bound checking and identifier matching to

force a complete memory safety [122, 123]. Cling enforces the temporal memory safety through type-safe memory reuse [17]. 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 [159].

5.5.3.2 Data-Flow Integrity

Data-flow integrity (DFI) generates the static data-flow graph (DFG) before the program execution [50]. 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 [154]. 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 [154] may work on DOP, as it protects some pieces of data, but not a panacea for all data.

5.5.3.3 Fine-grained Data-Plane Randomization

We have shown in Section 5.4.3 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 [31, 55]. 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-data and non-control-data) should be randomized frequently. A data-plane randomization with high performance and strong security guarantee is still an open question.

5.5.3.4 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 [56]. 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.

5.6 Related Work

In Section 5.5, we have discussed potential defenses against control-hijacking attacks and data-oriented attacks. In this section, we focus on the most closely related work.

Data-oriented attacks. In Section 5.1.1, we have an in-depth discussion on data-oriented attacks, including Chen *et al.* [56], control-flow bending attack [47] and our previous work FLOWSTITCH. The difference between DOP and others is that DOP does not rely on any specific security-critical data or functions, like password 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 security-critical data at particular program locations, like system call entry. Such defenses usually have acceptable overhead as critical code is rarely invoked in the real execution. But protecting all data pointers for all instructions will introduce extremely high performance overhead.

Return-Oriented Programming. ROP technique and its variants have been extensively explored recently [33, 35, 37, 52, 102, 143, 149, 153]. For example, counterfeit object oriented programming (COOP) demonstrates that Turing-complete attacks can be built with only virtual function calls in C++ [143]. However, ROP attacks change

the control flow of the vulnerable program, which can be mitigated by rapidly advancing control-flow integrity solutions [15, 63, 129, 130, 160, 163, 182]. 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 [150], which is used by DOP to build further attacks. Other weird machines can be built with DWARF (Debugging With Attribute Records Format) bytecode [131], the page fault handling mechanism [27] or the DMA (direct memory access) component [140]. DOP demonstrates the Turing-completeness in the data plane of arbitrary x86 programs.

5.7 Summary

In this work, we show that with a single memory error, data-oriented attacks can mount Turing-complete computation 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.

Chapter 6

Conclusion

Memory errors are persistent threats to system security. The arms race between attackers and defenders will not end in the near future. Attackers are proactively seeking new exploit methods to bypass existing defense mechanisms. Even known vulnerabilities can cause severe damage with advanced exploits.

To address this problem, we propose to use systematic analysis to identify memory errors and discover new exploit methods. We define and work out three novel projects. We first provide a systematic solution to detect memory errors during the privilege-based isolation. After privilege-based isolation, program partitions get extra memory access capability, which may bring new memory errors into existing programs. Our detection method identifies real threats in real-world programs. Then we explore the new exploit methods in the data space. We propose a novel method, data-flow stitching, to systematically build data-oriented attacks. Instead of corrupting particular security-sensitive data, this method selects data flows as targets and stitch two disjoint ones to mount severe attacks. Data-oriented exploits can bypass most of the known defense mechanism, like DEP, CFI and even ASLR. Further, we systematically study the expressiveness of data-oriented attacks, with respect to the ability for arbitrary computations. We propose data-oriented programming (DOP) to systematically find all necessary elements to build expressive attacks. With a comprehensive evaluation, we demonstrate that Turing-complete data-oriented attacks can be built from common memory errors.

Our work provides a systematic solution to proactively analyze the program to detect memory errors and understand new exploits in the data space. It explores the new exploit directions, demonstrates the severity, and illustrates the urgency for new defense

mechanisms. We release the set of data-oriented exploits built by FLOWSTICH and data-oriented programming at *http://huhong-nus.github.io/advanced-DOP/*. As we know, this is the first exploit set for data-oriented attacks.

Bibliography

[1] Bitcoin - Open source P2P money. https://bitcoin.org/.

[2] BusyBox. http://www.busybox.net/.

[3] Endspent(3C). https://docs.oracle.com/cd/E36784_01/html/E36874/endspent-3c.html.

[4] Glibc - Gnu. https://www.gnu.org/s/libc/.

[5] How Effective is ASLR on Linux Systems? http://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/.

[6] MCrypt. http://mcrypt.sourceforge.net/.

[7] Musl libc. http://www.musl-libc.org/.

[8] OrzHTTPd. https://code.google.com/p/orzhttpd/.

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

[10] SOP Bypass Demos. https://goo.gl/4PfXEg.

[11] SSH Communications Security. http://www.ssh.com/.

[12] Sudo Main Page. http://www.sudo.ws/.

[13] The Heartbleed Bug. http://heartbleed.com/.

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

[15] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[16] Jonathan Afek and Adi Sharabani. Dangling pointer: Smashing the pointer for fun and profit. BlackHat USA, 2007.

[17] Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[18] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[19] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven Compositional Symbolic Execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[20] Bertrand Anckaert, Mariusz Jakubowski, Ramarathnam Venkatesan, and Koen De Bosschere. Run-time Randomization to Mitigate Tampering. In *Proceedings of the 2nd International Conference on Advances in Information and Computer Security*, 2007.

[21] Starr Andersen and Vincent Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory protection technologies, Data Execution Prevention. Microsoft TechNet Library, September 2004.

[22] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-Independent Sandboxing of Just-In-Time Compilation and Self-Modifying Code. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2011.

[23] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, 2011.

[24] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[25] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[26] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[27] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. The Page-fault Weird Machine: Lessons in Instruction-less Computation. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, 2013.

[28] Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, 2008.

[29] Daniel J. Bernstein. Some Thoughts on Security After Ten Years of Qmail 1.0. In *Proceedings of the 14th ACM Workshop on Computer Security Architecture*, 2007.

[30] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[31] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[32] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Pro-*

ceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015.

[33] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[34] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[35] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai 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.

[36] Richard Blum. Postfix. `http://www.postfix.org/`, 2001.

[37] Erik Bosman and Herbert Bos. Framing Signals - A Return to Portable Shellcode. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[38] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *Proceedings of the 16th USENIX Security Symposium*, 2007.

[39] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011.

[40] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, 2006.

[41] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, 2008.

[42] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[43] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[44] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[45] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[46] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[47] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[48] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[49] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. HI-CFG: Construction by Binary Analysis, and Application to Attack Polymorphism. In *Proceedgins of 18th European Symposium on Research in Computer Security*, 2013.

[50] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[51] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[52] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[53] Stephen Checkoway, Ariel J Feldman, Brian Kantor, J Alex Halderman, Edward W Felten, and Hovav Shacham. Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage. In *Proceedings of the 2009 Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, 2009.

[54] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[55] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. A Practical Approach for Adaptive Data Structure Layout Randomization. In *Proceedings of the 20th European Symposium on Research in Computer Security*, 2015.

[56] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[57] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[58] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, 2014.

[59] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2011.

[60] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[61] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, 2000.

[62] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[63] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[64] CVE. Adobe Flash Player Type Confusion (CVE-2015-0336). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0336`.

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

[66] CVE. Buffer underflow in the ssl_decrypt_record function in Wireshark (CVE-2015-0564). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0564.

[67] CVE. Double free vulnerability in win32k.sys in the kernel-mode drivers in Microsoft Windows 8.1 (CVE-2-15-0058). http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-0058.

[68] CVE. Ghttpd 1.4 Daemon Buffer Overflow Vulnerability (CVE-2001-0820). https://cve.mitre.org/cgi-bin/cvename.cgi?name=2001-0820.

[69] CVE. Heap-Based Buffer Overflow in Gethostbyname (CVE-2015-0235). https://access.redhat.com/articles/1332213.

[70] CVE. Heap-based buffer overflow in Null HTTP Server (CVE-2002-1496). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1496.

[71] CVE. HTTPDX tolog() Function Format String Vulnerability (CVE-2009-4769). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-4769.

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

[73] CVE. Mozilla Firefox ”dangling pointer” Vulnerability (CVE-2011-0073). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0073.

[74] CVE. Nginx: Stack-based Buffer Overflow (CVE-2013-2028). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028.

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

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

[77] CVE. Sudo Format String Vulnerability (CVE-2012-0809). http://cve.mitre.org/cgi-bin/cvename.cgi?name=2012-0809.

[78] CVE. Use-After-Free Vulnerability in Proftpd (CVE-2010-4130). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-4130.

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

[80] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[81] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[82] Gregory J. Duck and Roland H. C. Yap. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[83] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic Spyware Analysis. In *Proceedings of USENIX Annual Technical Conference*, 2007.

[84] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[85] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[86] Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *Proceedings of the workshop on New security paradigms*, 2000.

[87] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, May 2015.

[88] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[89] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of USENIX Annual Technical Conference*, 2008.

[90] Aurélien Francillon and Claude Castelluccia. Code Injection Attacks on Harvard-architecture Devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[91] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, 2007.

[92] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[93] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[94] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, 2008.

[95] Patrice Godefroid and Ankur Taly. Automated Synthesis of Symbolic Instruction Encodings from I/O Samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[96] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size Does Matter: Why Using Gadget-chain Length to Prevent Code-reuse Attacks is Hard. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[97] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[98] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately Choosing Execution Runs for Software Fault Localization. In *Proceedings of the 15th International Conference on Compiler Construction*, 2006.

[99] Sean Heelan. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities. Technical report, Computing Laboratory, University of Oxford, September 2009.

[100] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'D My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[101] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[102] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size Does Matter in Turing-complete Return-oriented Pro-

gramming. In *Proceedings of the 6th USENIX Conference on Offensive Technologies*, 2012.

[103] Allen D Householder. Well There's Your Problem: Isolating the Crash-Inducing Bits in a Fuzzed File. Technical report, DTIC Document, 2012.

[104] Intel. Intel 64 and IA-32 architectures software developers manual - Chapter 28 VMX support for address translation. http://www.intel.com/content/dam/www/public/us/en/documents/ manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[105] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, 2002.

[106] Douglas Kilpatrick. Privman: A Library for Partitioning Applications. In *Proceedings of USENIX Annual Technical Conference*, 2003.

[107] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7), July 1976.

[108] Tim Kornau. Return Oriented Programming for the ARM Architecture. *Master's thesis, Ruhr-Universitat Bochum*, 2010.

[109] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[110] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[111] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[112] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed Symbolic Execution. In *Proceedings of the 18th International Conference on Static Analysis*, 2011.

[113] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[114] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[115] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for Tcb Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2008.

[116] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[117] Ziyuan Meng and Geoffrey Smith. Calculating Bounds on Information Leakage Using Two-bit Patterns. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, 2011.

[118] Microsoft. _set_printf_count_output. `https://msdn.microsoft.com/en-us/library/ms175782.aspx`.

[119] Daniel Moghimi. Subverting without EIP. `http://www.moghimi.org/subverting-without-eip/`, 2014.

[120] David Alexander Molnar, David Molnar, David Wagner, and David Wagner. Catchconv: Symbolic Execution and Run-Time Type Inference for Integer Conversion Errors. Technical report, UC Berkeley EECS, 2007.

[121] MicroSoft MSDN. Control Flow Guard. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx, 2016.

[122] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve 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.

[123] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 9th International Symposium on Memory Management*, 2010.

[124] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[125] Nergal. The Advanced Return-Into-Lib(c) Exploits. http://phrack.com/issues.html?issue=67&id=8, November 2007.

[126] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[127] James Newsome, Stephen McCamant, and Dawn Song. Measuring Channel Capacity to Distinguish Undue Influence. In *Proceedings of the ACM SIGPLAN 4th Workshop on Programming Languages and Analysis for Security*, 2009.

[128] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[129] Ben Niu and Gang Tan. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[130] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[131] James Oakley and Sergey 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.

[132] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49), 1996.

[133] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

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

[135] Mathias Payer and Thomas R. Gross. String Oriented Programming: When ASLR is Not Enough. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[136] Captain Planet. A Eulogy for Format Strings. `http://phrack.org/issues/67/9.html`.

[137] Dan R. K. Ports and Tal Garfinkel. Towards Application Security on Untrusted Operating Systems. In *Proceedings of the 3rd Conference on Hot Topics in Security*, 2008.

[138] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[139] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An Approach for Debugging Evolving Programs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009.

[140] Michael Rushanan and Stephen Checkoway. Run-DMA. In *Proceedings of the 9th USENIX Conference on Offensive Technologies*, 2015.

[141] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, 2004.

[142] Jonathan Salwan. ROPgadget - Gadgets Finder and Auto-Roper. `http://shell-storm.org/project/ROPgadget/`.

[143] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten 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.

[144] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[145] SecurityFocus. Musl libc inet_pton.c Remote Stack Buffer Overflow Vulnerability (CVE-2015-1817). `http://www.securityfocus.com/bid/73408`.

[146] SecurityFocus. OrzHTTPd Remote Format String Vulnerability (bugtraq ID: 41956). `http://www.securityfocus.com/bid/41956/info`.

[147] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[148] Fermin J Serna. The Info Leak Era on Software Exploitation. *Black Hat USA*, 2012.

[149] Hovav 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.

[150] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. “Weird Machines” in ELF: A Spotlight on the Underappreciated Metadata. In *Proceedings of the 7th USENIX Conference on Offensive Technologies*, 2013.

[151] Shweta Shinde, Shruti Tople, Deepak Kathayat, and Prateek Saxena. PODARCH: Protecting Legacy Applications with a Purely Hardware TCB. Technical Report NUS-SL-TR-15-01, School of Computing, National University of Singapore, February 2015.

[152] Christopher Small. A Tool for Constructing Safe Extensible C++ Systems. In *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies*, 1997.

[153] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza 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*, 2013.

[154] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 23th Annual Network and Distributed System Security Symposium*, 2016.

[155] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[156] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[157] Matt Staats and Corina Păsăreanu. Parallel Symbolic Execution for Structural Test Generation. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010.

[158] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[159] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[160] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[161] Tutorial, Heap Spray Exploit. Internet explorer use after free aurora vulnerability. `http://grey-corner.blogspot.com/2010/01/heap-spray-exploit-tutorial-internet.html`.

[162] Ubuntu. List of Programs Built with PIE, May 2012. `https://wiki.ubuntu.com/Security/Features#pie`.

[163] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[164] Perry Wagle and Crispin Cowan. Stackguard: Simple Stack Smash Protection for GCC. In *Proceedings of the GCC Developers Summit*, 2003.

[165] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM symposium on Operating systems principles*, 1993.

[166] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[167] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[168] Yongzheng Wu, Sai Sathyanarayan, Roland H. C. Yap, and Zhenkai Liang. Codejail: Application-Transparent Isolation of Libraries with Tight Program Interactions. In *Proceedings of 17th European Symposium on Research in Computer Security*, 2012.

[169] Xiaobo Chen. ASLR Bypass Apocalypse in Recent Zero-Day Exploits. `https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html`, 2014.

[170] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[171] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[172] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[173] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving Application Security with Data Flow Assertions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[174] Yves Younan. FreeSentry: Protecting against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, 2015.

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

[176] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, 2007.

[177] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A Retargetable Framework for Low-level Inlined-reference Monitors. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[178] Bin Zeng, Gang Tan, and Greg Morrisett. Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM conference on Computer and Communications Security*, 2011.

[179] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, 2015.

[180] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[181] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[182] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[183] Mu Zhang and Heng Yin. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21st Network and Distributed System Security Symposium*, 2014.