Wright State University

CORE Scholar

Browse all Theses and Dissertations | Theses and Dissertations

2018

Verifying Data-Oriented Gadgets in Binary Programs to Build Data-Only Exploits

Zachary David Sisco
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

 Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation

Sisco, Zachary David, "Verifying Data-Oriented Gadgets in Binary Programs to Build Data-Only Exploits" (2018). *Browse all Theses and Dissertations*. 1998.
https://corescholar.libraries.wright.edu/etd_all/1998

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.

VERIFYING DATA-ORIENTED GADGETS IN BINARY PROGRAMS TO BUILD DATA-ONLY EXPLOITS

A thesis submitted in partial fulfillment of the
requirements for the degree of
Master of Science

By

ZACHARY DAVID SISCO
B.S., Ohio University, 2014

2018
Wright State University

WRIGHT STATE UNIVERSITY

GRADUATE SCHOOL

June 14, 2018

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Zachary David Sisco ENTITLED Verifying Data-Oriented Gadgets in Binary Programs to Build Data-only Exploits BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science.

Adam R. Bryant, Ph.D.
Thesis Co-Director

John M. Emmert, Ph.D.
Thesis Co-Director

Mateen M. Rizki, Ph.D.
Chair, Computer Science and Engineering

Committee on Final Examination:

Meilin Liu, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Barry Milligan, Ph.D.
Interim Dean of the Graduate School

Abstract

Sisco, Zachary David. M.S. Computer Science and Engineering, Wright State University, 2018. Verifying Data-Oriented Gadgets in Binary Programs to Build Data-Only Exploits.

Data-Oriented Programming (DOP) is a data-only code-reuse exploit technique that "stitches" together sequences of instructions to alter a program's data flow to cause harm. DOP attacks are difficult to mitigate because they respect the legitimate control flow of a program and by-pass memory protection schemes such as Address Space Layout Randomization, Data Execution Prevention, and Control Flow Integrity. Techniques that describe how to build DOP payloads rely on a program's source code. This research explores the feasibility of constructing DOP exploits without source code—that is, using only binary representations of programs. The lack of semantic and type information introduces difficulties in identifying data-oriented gadgets and their properties. This research uses binary program analysis techniques and formal methods to identify and verify data-oriented gadgets, and determine if they are reachable and executable from a given memory corruption vulnerability. This information guides the construction of DOP attacks without the need for source code, showing that common-off-the-shelf programs are also vulnerable to this class of exploit.

Contents

List of Figures

List of Tables

Acknowledgements

The research presented in this thesis was funded by Edaptive Computing Inc. through Air Force Contract FA8650-14-D-1724-0003.

Chapter 1

Introduction

"Data-only" attacks are a class of exploit triggered by a memory corruption vulnerability that manipulate a program's data plane. Instead of hijacking control flow by manipulating return addresses and function pointers, these attacks cause harm by changing a program's logic and decision-making routines (Chen et al. 2005). Because data-only attacks respect a program's inherent control flow, they are harder to mitigate than control-flow hijacking exploits using current defense mechanisms like Address Space Layout Randomization, Data Execution Prevention, and Control-flow Integrity.

Data-oriented programming is a data-only code-reuse exploit technique that stitches together sequences of data-oriented instructions to simulate computation (Hu et al. 2016). In contrast to code-reuse attacks like return-oriented programming (Shacham 2007), data-oriented programming causes harm through manipulating a program's data plane while preserving the integrity of its control flow. Chaining together sequences of instructions that simulate common micro-operations—such as assignment, arithmetic, and conditionals—enable attackers to craft expressive, even Turing-complete, exploits (Hu et al. 2016). These sequences of instructions are called *data-oriented gadgets*. Correctly classifying data-oriented gadgets and their properties is

critical for successfully carrying out a data-oriented programming attack.

This thesis presents a methodology for classifying data-oriented gadgets in general binary programs without source code. The classification methodology uses data-flow analysis and program verification techniques to identify and verify data-oriented gadgets and their properties. Current techniques rely on source code to classify data-oriented gadgets. By classifying data-oriented gadgets without source code, the methodology and resulting prototype presented in this thesis expand the range of software that can be analyzed for this kind of threat—including "common-off-the-shelf" binaries, closed-source binaries, and legacy programs. This enables security analysts to investigate a generic binary and determine if any present data-oriented gadgets can be triggered from a given vulnerability. Through this, the prototype developed in this research demonstrates the feasibility of crafting data-oriented exploits in binaries without source code. Furthermore, this research explores the differences in classifying data-oriented gadgets with and without source code and how compilers introduce differences in the kinds of gadgets available in a binary and how they are discovered.

This thesis is organized as follows. Chapter 2 presents an overview of data-only attacks and data-oriented programming. Chapter 3 presents the data-oriented gadget classification methodology for binary programs. Chapter 4 presents data-oriented gadget classification results across a suite of programs and evaluates classification differences between binary and source-based analysis techniques as well as differences between compilers. Chapter 5 concludes this thesis with discussion of the results, related work, and future work.

Chapter 2

Background

Malicious entities exploit memory corruption vulnerabilities in software to do harm. These include errors such as stack and heap buffer overflows, integer overflows, use-after-free, double-free and format string vulnerabilities (Chen et al. 2005). These exploits alter a program's control data—such as return addresses or function pointers—in order to inject malicious code or reuse library code (in the case of return-oriented programming (Shacham 2007)) to cause harm. Thus, defense mechanisms that mitigate these attacks focus on protecting a program's *control data*.

Stack canaries (Cowan et al. 1998) prevent control data from being overwritten by inserting randomized "canary" values in between local variables and control data on the stack. "W⊕X" (Pax Team 2003*b*), a protection scheme on Linux and BSD-based operating systems, ensures that no memory region is marked both writable ('W') and executable ('X') at the same time. A similar scheme for Windows operating systems, Data Execution Prevention (DEP), marks data regions in memory as non-executable (Andersen & Abella 2004).

Address Space Layout Randomization (ASLR) (Pax Team 2003*a*) randomizes the locations of sections in an executing program—such as the stack, heap, and libraries—in order to prevent code reuse attacks like return-oriented programming (Shacham

2007) and its variants (Checkoway et al. 2010, Bletsch et al. 2011, Bittau et al. 2014, Bosman & Bos 2014, Carlini & Wagner 2014, Göktas et al. 2014, Schuster et al. 2015, Hu et al. 2016). These attacks depend on knowing the addresses of libraries and program sections. Therefore, randomizing the addresses during execution makes it harder for the attacks to succeed. Program shepherding (Kiriansky et al. 2002) and Control-flow Integrity (Abadi et al. 2005) are methods that ensure a program follows its control-flow graph during execution, thus thwarting any attacks that hijack control from a program. However, these control-oriented defense mechanisms do not mitigate *data-only* attacks (Chen et al. 2005)—also called non-control data attacks, or data-oriented attacks.

Data-only attacks differ from control-data attacks in that they do not alter the control flow of a program. Rather than alter return addresses and function pointers, a data-only attack tampers with data that affects the program's logic or decision-making routines (Chen et al. 2005). Such security-critical non-control data includes:

- *Configuration Data* (Chen et al. 2005)—loaded by a program at runtime which initialize data structures that control a program's behavior. Configuration files may also define access control policies and set file path directives that determine the location of other executables at runtime. Corrupting configuration data may change the program's behavior and overwrite access control policies.

- *User Identity Data* (Chen et al. 2005)—includes user ID, group membership, and access permissions. Corruption of this data may allow an attacker to perform unauthorized actions.

- *User Input Data* (Chen et al. 2005)—an attacker may exploit a program's input validation methods by providing valid user input, then later corrupting it after data validation to launch an attack. This is known as a "Time of Check to Time of Use" attack.

- *Decision-Making Data* (Chen et al. 2005)—logical expressions determine how a program's control flow branches. Corrupting the values used in these expressions, or the final boolean result, can change critical paths taken by a program.

- *Passwords and Private Keys* (Hu et al. 2015)—leaking critical security information helps an attacker bypass access controls.

- *Randomized Values* (Hu et al. 2015)—memory protection schemes such as stack canaries and ASLR use randomization. Understanding the randomization strategies used at runtime helps an attacker bypass these defenses.

- *System Call Parameters* (Hu et al. 2015)—corrupting these parameters allows an attacker to change a program's behavior for privileged operations (e.g., `setuid()`).

Exploiting these types of security-critical data cause harm in the form of sensitive information leakage, privilege escalation, and arbitrary code execution. The OpenSSL *Heartbleed* vulnerability (US-CERT 2014) is an example of a data-oriented exploit that leaks sensitive data—including private keys—without subverting the program's control flow. Due to a missing bounds check in the OpenSSL heartbeat request and response protocol, an attacker sends a legitimate payload with a specified length up to 64 kilobytes larger than the payload. Since the length field is not verified against the actual length of the payload, memory leakage is caused by copying the response into a buffer larger than the payload.

Another example of a data-only attack is found in a format string error in `wu-ftpd` (version 2.6.0), a free FTP server daemon. A snippet of the vulnerable source code is shown in Listing 2.1. The attack exploits the format string vulnerability in line 5 to overwrite security-critical *user identity data* `pw->pw_uid` with 0—the root user's ID (Chen et al. 2005). Then, line 7 temporarily escalates to root privileges in order to invoke `setsockopt()` (Chen et al. 2005). Line 10 intends to drop root user privileges

but due to the overwritten data from the format string error, instead retains root user privileges. This demonstrates root privilege escalation without overwriting return addresses or function pointers (Chen et al. 2005).

```
struct passwd { uid_t pw_uid; ... } *pw;
...
int uid = getuid();
pw->pw_uid = uid;
printf(...); // format string vulnerability
...
seteuid(0); // set root id
setsockopt(...);
...
seteuid(pw->pw_uid); // set unprivileged user id
...
```

Listing 2.1: Vulnerable code snippet in `wu-ftpd`.

2.1 Data-Oriented Programming

A general method for constructing data-only attacks is called *data-oriented programming* (DOP) (Hu et al. 2016). Given a vulnerable program, DOP builds Turing-complete data-only attacks capable of a high degree of expressiveness and arbitrary computation (Hu et al. 2016). The methodology resembles return-oriented programming (Shacham 2007) and its variants (Checkoway et al. 2010, Bletsch et al. 2011, Bittau et al. 2014, Bosman & Bos 2014, Carlini & Wagner 2014, Göktas et al. 2014, Schuster et al. 2015), where data-oriented programming uses *data-oriented* gadgets to build exploits. The distinction from these techniques is that *data-oriented* gadgets do not violate a program's legitimate control flow.

Data-oriented gadgets simulate a Turing machine by forming micro-operations such as load, store, jump, arithmetic and logical calculations. These gadgets are built from short sequences of instructions in a vulnerable program. These are different from code gadgets in return-oriented programming because data-oriented gadgets

must execute in a legitimate control flow (Hu et al. 2016). Additionally, data-oriented gadgets persist the output of their operations only to memory—whereas code gadgets in return-oriented programming can use memory or registers (Hu et al. 2016). Overall the requirements for building valid data-oriented exploits are stricter, but one benefit is that data-oriented gadgets are not required to execute one after another; they can be spread across functions or blocks of code.

A *gadget dispatcher* chains and sequences a series of data-oriented gadgets to form an attack (Hu et al. 2016). This is also constructed from short sequences of instructions. The most common code sequence for a dispatcher is a loop—allowing attackers to select and repeatedly invoke gadgets each iteration. Attackers control the selection and activation of gadgets through the program's memory error (Hu et al. 2016). The selection of gadgets and the termination of the loop is either encoded in a single payload, or *interactively* manipulated by the attacker from repeated memory corruptions at the start of each iteration.

Overall, the evaluation of data-oriented programming by Hu et al. (2016) shows that data-oriented gadgets are as prevalent in software as return-oriented gadgets and it is possible to construct Turing-complete exploits that bypass current memory protections.

2.2 Data-Oriented Programming Without Source

This thesis explores the feasibility of constructing data-oriented programming exploits in binary programs without source code. Current techniques rely on a program's source code for semantic and type information to classify data-oriented gadgets. This information is not available in binary programs.

Correctly classifying gadgets is necessary for constructing DOP exploits. To stitch together a sequence of data-oriented gadgets an attacker tracks two aspects of every

gadget: (1) the semantics of the gadget (the micro-operation it simulates), and (2) the parameters under control of the gadget. These aspects encompass correct classification. Thus, a methodology that achieves this without source code expands the range of programs that can be analyzed for this class of exploit. This includes generic binaries, "common-off-the-shelf" binaries, closed-source binaries, and legacy programs. This capability is currently not available and achieving it enables security analysts to determine the kinds of data-oriented gadgets present in a general binary and if they can be triggered from a given vulnerability.

Chapter 3

Methodology

There are three phases to classifying data-oriented gadgets in binary programs:

1. Identify potential gadgets using data-flow analysis techniques (Section 3.1);
2. Determine the semantics of the gadgets using program verification techniques (Section 3.2);
3. Given a dynamic function trace triggering a vulnerable function in the program, determine the reachability of the gadgets to the vulnerable program point (Section 3.6).

The first two phases are the focus of this work, as the third phase, reachability, follows immediately from phase one and two.

As defined by Hu et al. (2016), a data-oriented gadget is a sequence of instructions beginning with a load and ending with a store. The instructions in between determine the semantics of the gadget. Hu et al. (2016) defines a basic language to express data-oriented gadgets, MinDOP (Table 3.1). MinDOP defines expressions for assignment, dereference (load and store), arithmetic, logical, and comparison operations. To carry out an attacker's payload, MinDOP expresses a virtual instruction set that manipulates virtual registers to simulate computation.

Semantics	C Instructions	DOP Virtual Instructions
Binary operation	`a ◇ b`	`*p ◇ *q`
Assignment	`a = b`	`*p = *q`
Load	`a = *b`	`*p = **q`
Store	`*a = b`	`**p = *q`
Where `p = &a`; `q = &b`; and ◇ is arithmetic/logical/comparison operation.		

Table 3.1: MinDOP language by Hu et al. (2016), and how it relates to C instructions.

For example, Figure 3.1 shows a data-oriented gadget in C code and then its corresponding x86 assembly instructions. This is an addition gadget, adding the values of `*p` and `*q` and storing the result in `*p`. Lines 1–4 of the assembly instructions in Figure 3.1 load and dereference the values of `p` and `q`. Line 5 is the addition operation that makes this an addition gadget. The final instruction is a store instruction—making this a valid data-oriented gadget—storing the result of the addition operation in `*p`.

```
*p += *q; /* p, q are (int*) type */
```

```
1 mov eax, DWORD PTR [ebp-0xC]        ;load p to eax
2 mov edx, DWORD PTR [eax]            ;load *p to edx
3 mov eax, DWORD PTR [ebp-0x10]       ;load q to eax
4 mov eax, DWORD PTR [eax]            ;load *q to eax
5 add edx, eax                        ;add *q to *p
6 mov eax, DWORD PTR [ebp-0xC]        ;load p to eax
7 mov DWORD PTR [eax], edx            ;store edx in *p
```

Figure 3.1: Example showing a snippet of C code and the corresponding X86 assembly instructions.

3.1 Finding Potential Data-oriented Gadgets

To identify data-oriented gadgets (hereon referred to as "gadgets") in binary programs, we disassemble the binary and lift the instructions to an intermediate representation. To do this we use "angr" (Shoshitaishvili et al. 2016), a binary program analysis framework written in Python that utilizes static and concolic analysis tech-

niques. The intermediate representation angr uses is VEX-IR, which angr exposes via Python bindings (Shoshitaishvili et al. 2015). VEX-IR is an assembly-like intermediate representation used for binary analysis. It uses Static Single Assignment (SSA), explicitly tracks instruction side-effects, and abstracts away architectural differences to allow analysis for a variety of architectures.

Using angr's built-in ability to recover functions and loops from the program's control-flow graph, we scan each loop in each function of the binary. It is necessary to identify gadgets starting from loops, otherwise the gadgets will not be able to be invoked from a gadget dispatcher—essentially, a loop that allows for continued execution of a gadget. In addition to scanning the instructions of loops, we also follow any function calls within the loops and scan for gadgets in those functions. We consider these gadgets "reachable" from the original, enfolding loop.

To begin gadget identification, we consider any Store instruction and analyze its preceding statements. Because a Store instruction has two arguments, we trace the instructions for the variables of each argument. This tracing is done through backward static program slicing. Given a program P, a backward program slice at program point p with set of variables V contains only those preceding statements in P that affect the variables in V at p (Weiser 1981). The resulting program slice contains a subset of statements in P with only those statements that contribute to the values in V at p. Since data-oriented gadgets always end in with a Store instruction, a backward program slice begins at the program point of a Store instruction with relevant variables being the destination of the store and the value being stored.

Gadget identification starts at the basic block level. As a result, this reduces the complexity of program slicing because there are no loops or conditionals in a basic block. Thus, a static backward program slice begins with the algorithm in Listing 3.1. The pseudocode in Listing 3.1 describes how to set all relevant variables in a given basic block. Each statement in a basic block maps to a set of relevant variables.

Initially, only the Store statement has relevant variables set—these are the value and destination arguments. Then, the algorithm propagates relevance by considering a statement and its successor statement. If the variable defined by the statement is in not the set of relevant variables for the successor statement then that variable is added to the successor statement's relevant variables. If the statement does define a variable that is in the set of relevant variables for the successor statement, then the algorithm adds all variables used by the current statement into its own set of relevant variables. The result of this algorithm is a mapping from each statement in the basic block to a set of variables indicating that those variables contribute in some way to that statement.

```
function: SetRelevantVariables(B, relevantVariables)
input: B, basic block; relevantVariables, maps a statement to a set of variables
output: relevantVariables

foreach statement i and successor statement j in B:
    if i.LHS in relevantVariable[j] then:
        // add all variables used by i to the relevant variables of i
        relevantVariables[i].add(i.variables)
    else:
        // add that variable to the relevant variables of j
        relevantVariables[j].add(i.LHS)
```

Listing 3.1: Pseudocode for algorithm that sets relevant variables of a basic block. Note that i.LHS refers to the left-hand side of the statement i—that is, the variable being defined in an assignment statement. See Appendix A.1 for the Python source code implementation.

Note that the algorithms and formalisms in this methodology section correspond to Python source code for the implementation of a data-oriented gadget classification tool. The relevant source code is included in the Appendix. The algorithm in Listing 3.1 corresponds to Appendix A.1.

Then, Listing 3.2 builds the program slice by referencing the set of relevant variables. Again, considering each statement in the basic block and its successor, the algorithm checks if the variable defined in the current statement is in the set of relevant variables for the proceeding statement. If so, the algorithm adds the current statement to the program slice.

```
function: BackwardProgramSlice(B, relevantVariables)
input: B, basic block; relevantVariables, maps a statement to a set of variables
output: program slice, set of statements

relevantVariables ← SetRelevantVariables(B, relevantVariables)
foreach statement i and successor statement j in B:
    if i.LHS in relevantVariables[j] then:
        add i to the program slice
```

Listing 3.2: Pseudocode for algorithm that builds a backward program slice given a basic block. Note that i.LHS refers to the left-hand side of the statement i—that is, the variable being defined in an assignment statement. See Appendix A.2 for the Python source code implementation.

Then, given a program slice with respect to a Store instruction, we separate the program slice to select only the statements that are relevant to each argument of the Store. Listing 3.3 describes a backward data-flow analysis algorithm that does this given a target variable and program slice, returning a stack of relevant statements. The backward data-flow analysis algorithm traverses a program slice in reverse order looking for statements that define v, the target variable. Once found, the algorithm pushes the statement onto the output stack, then recursively calls itself for each of the variables in the right-hand side of the definition of v. This is repeated until the program slice is completely traversed. The backward data-flow analysis algorithm presented here is a variation of *liveness analysis*—that is, a variable x at program point p is *live* if the value of x at p could be used along some path starting at p (Aho et al. 2006). The difference here is that Listing 3.3 returns the path (sequences of instructions) that the target variable is live at.

To find a potential data-oriented gadget Listing 3.4 combines the algorithms in Listings 3.1–3.3. This starts from the basic block-level with a prospective Store instruction and uses each of the previous algorithms to generate two program slices that trace each of the arguments of the Store instruction. The resulting pair of instruction sequences is a potential data-oriented gadget.

The following example in Listing 3.5 demonstrates how backward static program slicing followed by the backward data-flow analysis algorithm produces two program

```
function: BDFA(v, slice, istack)
input: v, target variable; slice, program slice
output: istack, instruction stack tracing v

if slice is empty then:
    return
i ← slice.pop()
if i is Assignment Instruction then:
    if i.LHS = v then:
        istack.push(i)
        rhs ← GetVariables(i.RHS)
        foreach variable t in rhs:
            BDFA(t, slice, istack)
else:
    BDFA(v, slice, istack)
```

Listing 3.3: This pseudocode presents an algorithm for Backward Data-flow Analysis that picks out the statements in a program slice that contribute to a single target variable. Note that i.LHS and i.RHS refer to the left and right-hand sides of the statement i. See Appendix A.3 for the Python source code implementation.

```
function: GetGadget(store, B)
input: store, a Store Instruction; B, basic block
output: a pair of instruction stacks

relevantVariables ← ∅
addrInstr ← ∅
dataInstr ← ∅

relevantVariables[store].add(store.addr)
relevantVariables[store].add(store.data)
progSlice ← BackwardProgramSlice(B, relevantVariables)
addrInstr ← BDFA(store.addr, progSlice, addrInstr)
dataInstr ← BDFA(store.data, progSlice, dataInstr)
// Potential gadgets must have at least one Load instruction
if Load Instruction in addrInstr or dataInstr then:
    return ⟨addrInstr, dataInstr⟩
```

Listing 3.4: This pseudocode presents an algorithm for data-oriented gadget identification given a Store instruction in a basic block. Note that *store*.addr and *store*.data refer to the destination and value arguments of the Store instruction, respectively. See Appendix A.4 for the Python source code implementation.

slices that trace the definitions of the arguments to a Store instruction. In the example, `t37` and `t36` are the address and data variables for the Store instruction's arguments, respectively. The program slice for `t37` traces its definition to a load from register `EBP` (VEX-IR identifies this as offset 28). After adding an offset to the address of the base pointer, the next instruction loads the value and stores the result in `t33`. Then, the final instruction adds a constant value of `0x10` to `t33`, storing the final value in `t37`. Note how this program slice contains none of the instructions relevant to the definition of `t36`—only `t37`.

```
# Original basic block
t11 = GET:I32(offset=28) # 28 = EBP
t31 = Add32(t11,0xffffef70)
t33 = LDle:I32(t31)
t34 = Add32(t11,0xffffefec)
t36 = LDle:I32(t34)
t37 = Add32(t33,0x00000010)
STle(t37) = t36

# Program slice tracing t37
t11 = GET:I32(offset=28)
t31 = Add32(t11,0xffffef70)
t33 = LDle:I32(t31)
t37 = Add32(t33,0x00000010)

# Program slice tracing t36
t11 = GET:I32(offset=28)
t34 = Add32(t11,0xffffefec)
t36 = LDle:I32(t34)
```

Listing 3.5: Backward static program slice example in VEX-IR. The backward dataflow analysis algorithm in Listing 3.3 splits the basic block into two program slices for each argument to the Store instruction.

The following pseudocode in Listing 3.6 wraps all of the algorithms from Listings 3.1–3.4 together for whole-program potential gadget identification. Whole-program analysis starts from each function in the program, drilling down to each loop, and then to each basic block in the loop body. In addition to considering each Store instruction

in the loop body, the algorithm also checks function calls. Data-oriented gadgets in these function calls are also reachable from the original loop. The "followCallGraph()" function in Listing 3.6 traces the program's call graph from the loop body to the called function and returns the block of instructions corresponding to the called function.

```
function: GetPotentialGadgets(prog)
input: prog, program in VEX−IR
output: potentialGadgets, a list of pairs of instruction stacks

foreach func in prog:
    foreach loop in func:
        foreach basic block B in loop:
            foreach stmt in B:
                if stmt is Store Instruction then:
                    g ← getGadget(stmt, B)
                    potentialGadgets.add(g)
                if stmt is Call Instruction then:
                    target ← followCallGraph(stmt)
                    foreach stmt in target:
                        if stmt is Store Instruction then:
                            g ← getGadget(stmt, target)
                            potentialGadgets.add(g)
```

Listing 3.6: This pseudocode presents an algorithm for whole-program identification of potential data-oriented gadgets. See Appendix A.5 for the Python source code implementation.

Unlike Return-oriented Programming (ROP) gadgets—which end in a return instruction—data-oriented gadgets have two sequences of instructions to consider. This is due to the two arguments to the Store instruction. The identification algorithms presented here (Listings 3.1–3.4) describe how to build program slices that contain only the relevant statements for each variable argument in a given Store instruction. The next step is to identify the semantics of the instructions for each part of the gadget.

3.2 Program Verification Techniques to Classify Gadgets

Hu et al. (2016) classifies data-oriented gadget semantics using a heuristic algorithm. This favors speed over accuracy. However, in classifying gadgets in binary programs,

there is less semantic information available. This hinders the accuracy of a heuristic algorithm. Thus, using program verification techniques to verify the correctness of gadget semantics guards against misclassification. Additionally, for software security, this approach gives analysts a provably verified set of gadgets present in a binary.

This research follows the work of Schwartz et al. (2011) which uses program verification techniques to classify ROP gadgets in binary programs. The problem of classifying the semantics of a gadget involves considering a first-order predicate Q which describes the semantics of a gadget within a program S. If a gadget is of the type described by Q, then after executing the statements in S the program is in a state satisfying Q. To determine if Q can be satisfied we find the *Weakest Precondition* of S given Q—denoted $wp(S, Q)$. The weakest precondition, $wp(S, Q)$, is a predicate that characterizes all initial states of the program S such that it terminates in a final state satisfying Q—also called the *postcondition* (Dijkstra 1976).

Thus, gadget classification becomes a problem of deriving the weakest precondition of program slices. Characterized by Dijkstra (1976), *predicate transformers* are the rules that derive weakest preconditions from a program. Dijkstra's Guarded Command Language (GCL) is the syntax that encapsulates these transformations. Flanagan & Saxe (2001) adapted GCL and predicate transformers to derive verification conditions for Java programs. Brumley et al. (2007) adapted these rules again for use in binary analysis, which is the focus in this work.

Since data-oriented gadgets in binaries are limited to basic blocks, the semantic rules for computing the weakest precondition of a GCL-like program are reduced. This is because the potential instructions within the basic block of a gadget do not contain loops or conditional control-flow transfers.

Table 3.2 presents a GCL-like syntax for gadget verification. s ; s is composition of statements, that is, statements executed in sequence. $s \square s$ is the "choice" operation, representing a non-deterministic choice between the execution of two statements

(Flanagan & Saxe 2001). Although this application uses VEX-IR as the binary program intermediate representation, in general, any language can be used in its place. The core operations—assignment, load, store, arithmetic, logical, comparison—are common to intermediate languages.

GCL *Statement*	s	$::=$	$x := e$
		$\mid$	**assume** e
		$\mid$	$s \; ; \; s$
		$\mid$	$s \; \square \; s$
VEX-IR *Expression*	e	$::=$	t
		$\mid$	r
		$\mid$	m
		$\mid$	c
		$\mid$	$t \diamond t$
		$\mid$	$t \diamond c$
Operator	$\diamond$	$::=$	$+ \mid - \mid * \mid \div \mid \wedge \mid \vee \mid \oplus \mid$ $\ll \mid \gg \mid = \mid \neq \mid < \mid \leq$ $> \mid \geq \mid \%$
Assignment Value	x	$::=$	$t \mid r \mid m$
Temporary Variables	t	$\in$	$\mathcal{T}$
Registers File	r	$\in$	$\mathcal{R}$
Memory Array	m	$\in$	$\mathcal{M}$
Constants	c	$\in$	$\mathbb{Z}$

Table 3.2: A GCL-like syntax for specifying programs to apply predicate transformers to derive weakest preconditions.

To compute the weakest precondition of the instructions in a gadget, we first lift the VEX-IR statements to the GCL-like syntax in Table 3.2. Because the VEX-IR statements are in SSA form, contain no conditional control-flow transfers, and have been reduced to trace the effects of a single variable, the lifting process considers a smaller subset of possible instructions. These include assignment, load, store, and binary (arithmetic, logical, comparison) operations. Thus, translating to the GCL-

like syntax is trivial; for example:

```
        t2 = Load(0x880123)
        t1 = Add(t2, t3)
Store(t3) = t1
```

lifts to:

$$\mathcal{T}[2] := \mathcal{M}[\texttt{0x880123}];$$
$$\mathcal{T}[1] := \mathcal{T}[2] + \mathcal{T}[3];$$
$$\mathcal{M}[\mathcal{T}[3]] := \mathcal{T}[1].$$

There is one exception for comparison operations (such as $=$, $\neq$, $<$, and $\leq$). In this situation, both outcomes—true or false—need to be considered for computing the weakest precondition. Given a comparison function in VEX-IR, `Cmp()`, a VEX-IR statement `t1 = Cmp(t2, t3)` lifts to:

$$(\textbf{assume } (\mathcal{T}[2] \diamond_c \mathcal{T}[3]);\ \mathcal{T}[1] := 1;)\ \square$$
$$(\textbf{assume } \neg(\mathcal{T}[2] \diamond_c \mathcal{T}[3]);\ \mathcal{T}[1] := 0;),$$

where $\diamond_c$ is the corresponding comparison operator to `Cmp()`.

After lifting the gadget instructions to GCL, we apply the semantics in Figure 3.2. These are the predicate transformers as adapted by Brumley et al. (2007) for binary program analysis. See Appendix A.6 for the Python source code implementation which takes as input a sequence of statements in GCL syntax and postcondition, and derives the weakest precondition according to the rules in Figure 3.2.

$$\frac{}{wp(x := e, Q) : Q[e/x]}\text{WP-Assign}$$

$$\frac{}{wp(\textbf{assume } e, Q) : e \Rightarrow Q}\text{WP-Assume}$$

$$\frac{wp(s_2, Q) : Q_1 \quad wp(s_1, Q_1) : Q_2}{wp(s_1 \ ; \ s_2, Q) : Q_2}\text{WP-Sequence}$$

$$\frac{wp(s_1, Q) : Q_1 \quad wp(s_2, Q) : Q_2}{wp(s_1 \square s_2, Q) : Q_1 \wedge Q_2}\text{WP-Choice}$$

Figure 3.2: Semantics for deriving weakest precondition of a GCL-like program.

3.2.1 Example Arithmetic Gadget Classification

Listing 3.7 presents an example gadget performing an `ADD` operation with a constant. The VEX-IR statements in Listing 3.7 lift to the following GCL-like statements (abbreviated to $s_1; s_2; s_3$) in Equation 3.1.

$$\begin{aligned} s_1; s_2; s_3 = \\ \mathcal{T}[5] &:= \mathcal{M}[\texttt{0x805c7e8}]; \\ \mathcal{T}[1] &:= \mathcal{T}[5] + 2; \\ \mathcal{M}[\texttt{0x805c7e8}] &:= \mathcal{T}[1] \end{aligned} \tag{3.1}$$

```
t5 = LDle:I32(0x0805c7e8)
t1 = Add32(t5,0x00000002)
STle(0x0805c7e8) = t1
```

Listing 3.7: VEX-IR example program slice demonstrating an arithmetic `ADD` gadget.

For this to be a valid arithmetic gadget, we derive a valid weakest precondition for Equation 3.1 given a postcondition Q describing the semantics for an arithmetic gadget. In this case Q is ($\mathcal{M}[\texttt{0x805c7e8}] = \mathcal{T}[5] + 2$), as this describes the desired state of the program slice after the statements execute.

Figure 3.3 shows the application of the rules in Figure 3.2 on the gadget in Equation 3.1. The initial value of Q is $(\mathcal{M}[\texttt{0x805c7e8}] = \mathcal{T}[5] + 2)$. The resulting weakest precondition is a reflexive equality and is trivially valid, hence showing that the gadget is indeed a valid arithmetic gadget.

$$
\begin{aligned}
wp(s_1; s_2, wp(s_3, Q)) &= wp(s_1; s_2, \mathcal{T}[1] = \mathcal{T}[5] + 2) && \text{(WP-Assign)} \\
&= wp(s_1, wp(s_2, \mathcal{T}[1] = \mathcal{T}[5] + 2)) && \text{(WP-Sequence)} \\
&= wp(s_1, \mathcal{T}[5] + 2 = \mathcal{T}[5] + 2) && \text{(WP-Assign)} \\
&= (\mathcal{M}[\texttt{0x805c7e8}] + 2 = \mathcal{M}[\texttt{0x805c7e8}] + 2) && \text{(WP-Assign)}
\end{aligned}
$$

Figure 3.3: Application of weakest precondition derivation rules for example gadget in Equation 3.1.

To classify gadgets of different types, we specify the postconditions presented in Table 3.3. $\diamond_a$ is an arithmetic binary operator; $\diamond_\ell$ is a logical binary operator; and $\diamond_c$ is a comparison operator. **In** and **Out** represent parameters a gadget uses for the two arguments to the Store instruction—the destination and value, respectively. With the exception of Conditional or Comparison operations, these postconditions resemble the MinDOP syntax presented in Table 3.1.

Name	Parameters	Postcondition
Move	**Out**, **In**	**Out** = **In**
Load	**Out**, **In**	**Out** = $\mathcal{M}$[**In**]
Store	**Out**, **In**	$\mathcal{M}$[**Out**] = **In**
Arithmetic	**Out**, x, y	**Out** = $x \diamond_a y$
Logical	**Out**, x, y	**Out** = $x \diamond_\ell y$
Conditional	**Out**, x, y	$((x \diamond_c y) \Rightarrow \textbf{Out} = 1) \wedge$ $(\neg(x \diamond_c y) \Rightarrow \textbf{Out} = 0)$

Table 3.3: Postconditions for verifying data-oriented gadget semantics.

3.3 Scope Inference and Optimizations for Classifying Gadgets

Note that pointer information for the inputs is not included in Table 3.3. Since data-oriented programming treats memory as virtual registers to carry out computation, parameters **In** and **Out** must be pointers. Additionally, and in contrast to previous work, this research deals with binary programs without source code. Thus, variable information is not readily available and must be inferred. We gather this information before deriving the weakest precondition through a forward pass through the gadget's instructions. Not only does this provide pointer dereferencing information, but it also narrows the possible semantics that the gadget needs to be tested for, thus optimizing the implementation.

The forward pass looks for assembly conventions using disassembly data or architecture information provided by angr. With this, the forward pass identifies loads from the base pointer or other argument registers (dependent on architecture). Then, if the loaded value either loads again (dereferenced) or adds an offset and then loads again, the variable is a potential "virtual register" for a DOP program.

The forward pass also infers variable scope information. If a Load instruction uses a constant to load an address, the pass checks if the address falls within the `bss` or `data` sections of the binary file. If so, then the variable is global. If a variable is loaded from an address stored on the stack or in an argument register, the forward pass checks if the offset added to the variable is positive or negative. Based on architecture conventions, a positive or negative offset indicates the variable is either a function parameter or a local variable.

Additionally, the forward pass makes note of how many times each variable in a program slice is dereferenced. This information, combined with scope, provides details about a gadget to be able to stitch it together with other gadgets and allow

an attacker to perform arbitrary computation.

For example, the VEX-IR in Listing 3.8 presents two variables of interest—`t34` and `t38`. The forward pass scope inference algorithm determines that `t34` is a global variable because it loads from a memory address in the program's `data` section in Line 5. It also infers that `t38` is a local variable that's been dereferenced at least once. The forward pass determines this from Lines 1–4; here, the instructions add a negative offset to the address pointed to by the base pointer. Then, the value at the location is loaded, then loaded again. Through this inference process, the forward pass algorithm identifies the parameters for each potential gadget (as specified in Table 3.3) and prepares them for the verification step in Section 3.2.

```
1    t33 = GET:I32(offset=28) # 28 is EBP
2    t35 = Add32(t33, 0xffffffe0)
3    t37 = LDle:I32(t35)
4    t38 = LDle:I32(t37)
5    t34 = LDle:I32(0x805c7e8)
6    STle(t34) = t38
```

Listing 3.8: VEX-IR example program slice demonstrating two examples of variable scope inference in VEX-IR. `t34` is a global variable, and `t38` is a local variable.

3.4 Automating Gadget Classification and Verification

To automate gadget classification, we consider each potential gadget, run the forward pass to identify variables and their scopes in each program slice, compute the weakest precondition for each relevant gadget type and check the validity of the weakest precondition using the SMT solver Z3 (De Moura & Bjørner 2008). Thus, for a program slice S and postcondition Q, if the computed weakest precondition $wp(S, Q)$ is valid, then the gadget is verified to express the semantics defined in the postcondition Q.

This is repeated for each potential gadget found by the algorithm in Listing 3.6.

3.5 Identifying Gadget Dispatchers

Similar to Hu et al. (2016) we identify gadget dispatchers by finding data-oriented gadgets either within the bodies of loops or that are reachable from the body of a loop—that is, there is a path along the call graph from a function call in the loop body to the gadget. These loops are the dispatchers.

3.6 Reachability Analysis

The next step after identifying and classifying data-oriented gadgets is to determine their *reachability* from a vulnerable function. Since a DOP attack originates from a memory corruption, it is necessary that the gadgets used in the attack are reachable from that vulnerable function. We determine reachability in a manner similar to Hu et al. (2016) by capturing a dynamic function call trace of the program running with input that triggers the vulnerable function. Given the function call trace, we identify the functions invoked by the vulnerable function, and the loops surrounding the vulnerable function. We label the gadgets inside the invoked functions and enfolding loops as reachable from the dispatcher.

This completes all three phases of the methodology for classifying data-oriented gadgets in binaries as introduced at the beginning of this chapter. In total, this methodology describes a verified whole-program data-oriented gadget classification technique for general binaries. It can be applied to any architecture and requires no source code for analysis, utilizing data-flow analysis and program verification techniques to identify gadgets, verify their semantics, and determine if they can be triggered by vulnerable program points in a binary.

Chapter 4

Results

We implement the data-oriented gadget classification methodology for binary programs using Python 2.7.9 and the "angr" binary program analysis framework (Shoshitaishvili et al. 2016). The tool's name is DOGGIE—**D**ata-**O**riented **G**ad**g**et **I**dentifi**e**r. Please refer to the Appendix for the corresponding source code for key algorithms and formalisms. DOGGIE identifies and verifies the semantics of data-oriented gadgets in binary programs. The tool leverages the SMT solver Z3 (De Moura & Bjørner 2008) for verification. Once the tool classifies a data-oriented gadget it also reports the reachable loop, or dispatcher, from that gadget.

Additionally, the tool determines the *reachability* of gadgets to vulnerable functions in a binary program. This process first leverages Intel's Pin (Luk et al. 2005), a dynamic binary instrumentation tool, to capture a function trace of the target program executing with input that triggers a vulnerable function. Given such a function trace, DOGGIE labels the discovered gadgets that are invoked by the functions in the trace as reachable—meaning it is possible to trigger these gadgets from the vulnerable function.

The implementation of DOGGIE has one primary limitation with regards to gadget classification. DOGGIE does not verify gadgets that exhibit "complex" semantics.

We define "complex" as having more than two movement operations (assignment or dereference) and more than one binary operation (arithmetic, logical, or conditional). Although this implementation decision omits certain data-oriented gadgets, it is practical. Gadgets with long sequences of instructions performing multiple kinds of micro-operations are difficult to stitch together because there are more side effects to account for.

4.1 Evaluation

To evaluate how accurately DOGGIE classifies data-oriented gadgets in binary programs we compare the classification results of the tool with Hu et al. (2016)'s source-based gadget discovery tool. Hu et al. (2016)'s gadget discovery tool uses LLVM version 3.5.0 (Lattner & Adve 2004). We choose open-source programs for evaluation to compare results using both tools. The experimental setup consists of a host computer with an Intel x86 32-bit processor running Debian 8.10 on Linux kernel version 3.16. We compile each program using GCC 4.9.2 and Clang 3.5.0. The source-based tool from Hu et al. (2016) requires the programs to be compiled with Clang since it uses LLVM.

The selected programs include:

- curl—a tool that transfers data to or from a server using network protocols;
- imlib2—a graphics library for loading, saving, and rendering image files into different formats;
- libtiff—a library for reading and writing TIFF image files on Linux systems;
- nginx—an HTTP web server;
- optipng—a PNG file optimizer that compresses images;

- sudo—a system utility that allows users to run programs with elevated security privileges;
- unzip—a tool for extracting files from zip archives.

In addition to reporting classification results for data-oriented gadgets, the evaluation also reports gadget reachability for a given vulnerability. To do this we collect a function trace of the program running with a proof-of-concept exploit that triggers a disclosed vulnerability. The vulnerabilities for each program come from the CVE (Common Vulnerabilities and Exposures) database (CVE 2018). Because the source-based tool from Hu et al. (2016) does not report gadget reachability, we only provide reachability results of the binary programs using DOGGIE.

4.2 Classification Results

Table 4.1 presents the results of data-oriented gadget classification for binary and source-based programs using DOGGIE and Hu et al's LLVM pass, respectively. The table classifies gadgets according to two dimensions—semantics and scope. Semantics are the type of micro-operations that the gadgets simulate. Scope defines the context of the parameters for the gadget. For gadget scopes, 'G' is Global, 'H' is Hybrid (mixed between global and local), and 'L' is Local. For instance, a local gadget uses parameters that are locally scoped—modifications to these variables are limited to the scope of the function. A global gadget uses parameters that have global scope. An attacker can persist changes to these global variables and stitch gadgets together using the modified value of one gadget as input to a successive gadget. "Hybrid" scope gadgets consist of at least one local parameter and one global parameter. Additionally, each program has three entries—(1) the binary program compiled with GCC; (2) the binary program compiled with Clang; (3) and the source-code program compiled with Clang used with the LLVM pass by Hu et al. (2016).

Application	Version	Binary/Source	Compiler	Dispatchers	Assign			Deref			Arith			Logic			Cond		
					G	H	L	G	H	L	G	H	L	G	H	L	G	H	L
curl	7.41.0	B	GCC	71	99	143	27	62	25	559	36	303	16	5	0	0	1	0	0
		B	Clang	76	349	311	1	87	27	53	16	318	1	6	2	0	3	0	0
		S	Clang	11	0	2	15	0	2	26	2	0	5	1	2	17	0	0	4
imlib2	1.4.7	B	GCC	734	440	275	0	220	101	633	1208	256	137	65	19	15	4	4	0
		B	Clang	835	934	721	3	180	86	65	262	223	22	33	29	9	0	0	0
		S	Clang	152	1	5	55	25	183	94	12	96	390	3	219	411	3	1	19
libtiff	2.5.6	B	GCC	358	264	309	7	226	99	578	328	161	52	6	16	19	0	2	0
		B	Clang	374	451	290	35	87	175	5	148	179	5	15	11	1	1	0	0
		S	Clang	116	2	9	83	0	62	333	1	79	243	0	35	183	0	5	20
nginx	1.4.0	B	GCC	499	1316	526	1	333	190	87	276	440	9	97	2	4	3	0	0
		B	Clang	496	1104	497	11	378	328	114	225	426	14	83	11	3	4	0	0
		S	Clang	206	12	55	74	32	40	958	12	26	156	28	7	290	3	7	22
optipng	0.7.6	B	GCC	219	167	165	4	45	76	208	183	260	45	24	5	1	3	2	0
		B	Clang	249	244	114	13	19	42	38	144	235	8	23	1	1	1	0	0
		S	Clang	63	35	35	73	10	1	245	11	35	283	17	25	146	4	4	69
sudo	1.8.3p1	B	GCC	91	129	123	127	10	45	92	21	317	101	11	0	9	3	0	0
		B	Clang	65	103	44	1	13	15	12	14	239	0	12	0	0	3	0	0
		S	Clang	16	11	0	9	11	5	8	3	0	9	5	2	0	3	0	0
unzip	6.0	B	GCC	209	133	204	32	68	43	136	194	218	13	16	12	2	1	0	0
		B	Clang	182	215	32	0	21	11	12	161	139	2	8	0	3	1	0	0
		S	Clang	28	45	14	4	146	6	1	147	4	7	72	2	0	34	2	0

Table 4.1: Data-oriented gadget identification results.

The data in Table 4.1 demonstrates inconsistency in classifying data-oriented gadgets between programs with and without source code and between programs compiled with different compilers. With the exception of conditional gadgets in "sudo," there is no other case where gadget classification agrees in semantics or scope for the three cases (binary with GCC; binary with Clang; and source-code with Clang and LLVM).

For other cases such as "curl" conditional gadgets, the scope results between the binary and source versions do not match. The binary versions report global conditional gadgets with zero hybrid and local gadgets. The source version reports zero global conditional gadgets with only local gadgets. For each of the programs, despite the fact that Doggie omits classifying complex gadgets, it classifies more gadgets than the source-based analysis for a majority of Assignment, Dereference, and Arithmetic semantics.

If scope were removed from the data and Table 4.1 only presented total gadget numbers, the results would still vary in almost every case. Furthermore, there is as much of a difference between compilers as there is between binary and source-based analysis.

Table 4.2 presents statistics on discovered gadgets for each program. As Doggie and Hu et al's LLVM pass use different intermediate languages and operate exclusively

in either binary or source, Table 4.2 only reports statistics collected by DOGGIE for binary versions of the programs. Gadget length is the number of instructions in a gadget. Gadget parameters is the number of parameters the gadget controls.

Due to DOGGIE's limitations on gadget complexity, mean and median gadget parameter counts are two (with the exception of "optipng" compiled with GCC and "sudo" compiled with Clang, where mean is three). For a majority programs, the mean, median, and minimum statistics for each category between compilers are within ±3 of each other. This shows some consistency between compiled versions of a program. Maximum statistics tend to vary the most, especially for "gadgets per function" and "gadgets per dispatcher." Between compilers and programs, gadget length is consistent and varies by one or two—with GCC gadgets consistently being longer than Clang gadgets.

Application	Compiler	Gadget Length				Gadget Parameters				Gadgets per Function				Gadgets per Dispatcher			
		Mean	Median	Min	Max	Mean	Median	Min	Max	Mean	Median	Min	Max	Mean	Median	Min	Max
curl	GCC	7	7	3	18	2	2	2	4	9	1	4	553	27	4	1	672
	Clang	5	5	2	17	2	2	2	4	9	4	1	495	27	5	1	592
imlib2	GCC	7	7	4	29	2	2	2	5	11	4	1	223	12	7	1	527
	Clang	6	6	4	17	2	2	2	4	7	2	1	104	9	2	1	673
libtiff	GCC	8	7	4	23	2	2	2	5	7	3	1	257	9	3	1	539
	Clang	7	6	4	18	2	2	2	6	8	4	1	185	7	2	1	124
nginx	GCC	6	6	3	26	2	2	2	6	7	4	1	93	10	4	1	471
	Clang	6	6	3	26	2	2	2	6	7	4	1	107	9	3	1	448
optipng	GCC	7	7	4	33	3	2	2	9	7	4	1	96	11	3	1	254
	Clang	6	6	3	31	2	2	2	10	8	4	1	69	6	2	1	122
sudo	GCC	6	5	2	18	2	2	2	4	7	4	1	75	14	4	1	126
	Clang	5	5	2	14	3	3	2	4	5	4	1	48	11	5	1	58
unzip	GCC	7	6	4	28	2	2	2	7	10	4	1	81	12	2	1	296
	Clang	5	4	2	27	2	2	2	7	8	4	1	109	10	2	1	112

Table 4.2: Data-oriented gadget statistics collected by DOGGIE.

4.3 Reachability Results

Table 4.3 presents the results of data-oriented gadget reachability given a vulnerable function trace for binary programs using DOGGIE. Similar to the classification results table (Table 4.1), Table 4.3 organizes gadgets by semantics and scope. Additionally, for each program, to produce a vulnerable function trace we trigger a disclosed vulnerability. The table also lists the CVE number for the vulnerability. We only present

reachability results for binary programs using DOGGIE because the source-based tool from Hu et al. (2016) does not report reachability given a function trace.

Application	CVE	Compiler	Dispatchers	Assign			Deref			Arith			Logic			Cond		
				G	H	L	G	H	L	G	H	L	G	H	L	G	H	L
curl	2015-3144[1]	GCC	5	0	4	0	0	1	1	0	52	0	0	0	0	0	0	0
		Clang	8	0	4	0	0	0	2	0	56	0	0	0	0	0	0	0
imlib2	2016-3994[2]	GCC	12	7	6	0	7	3	17	3	36	0	0	0	4	0	0	0
		Clang	14	18	7	0	0	6	6	1	28	0	0	0	0	0	0	0
libtiff	2017-9935[3]	GCC	16	13	26	0	13	0	48	5	36	11	1	0	1	0	0	0
		Clang	14	16	9	0	5	3	2	7	9	0	2	0	0	0	0	0
nginx	2013-2028[4]	GCC	69	370	136	1	54	49	19	71	49	0	50	0	0	2	0	0
		Clang	77	297	110	0	80	51	18	44	44	0	28	3	1	1	0	0
optipng	2016-3982[5]	GCC	3	0	3	1	0	0	0	0	28	0	0	0	0	0	0	0
		Clang	4	0	0	0	0	0	0	0	24	0	0	0	0	0	0	0
sudo	2012-0809[6]	GCC	5	23	4	1	2	0	2	0	0	0	0	0	0	1	0	0
		Clang	9	20	0	0	1	0	2	1	0	0	12	0	0	0	0	0
unzip	2015-7696[7]	GCC	6	0	0	0	0	0	0	0	60	0	0	0	0	0	0	0
		Clang	6	0	0	0	0	0	0	0	48	0	0	0	0	0	0	0

Table 4.3: Data-oriented gadget reachability results with respect to a vulnerable function trace through a reported vulnerability from the CVE database (CVE 2018).

Again, the reachability results in Table 4.3 show a lack of consistency within programs and between compilers. Each program does not have the same reachable gadgets depending on the compiler. Additionally, not every classified gadget is reachable from the chosen vulnerability. Each program has reachable gadgets in at least one semantics category. "nginx" and "sudo" have reachable gadgets for all semantics. "curl," "imlib2," and "libtiff" at least have assignment, dereference, and arithmetic gadgets—which according to Hu et al. (2016) is sufficient for constructing Turing-complete DOP attacks. "nginx" reports the highest number of reachable gadgets compiled with either GCC and Clang. From these results, reachability depends closely on the vulnerability present in the program and the frequency and type of gadgets present.

[1]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3144
[2]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3994
[3]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9935
[4]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028
[5]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3982
[6]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0809
[7]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7696

4.4 Case Studies

We choose three programs to investigate classification and reachability results. These case studies detail the differences in data-oriented gadget classification between compilers and between binary and source-based program analysis. To demonstrate a variety of application types the case studies use a systems utility (sudo), a file decompression tool (unzip), and an HTTP server (nginx).

4.4.1 sudo

The system utility "sudo" demonstrates the differences between binary and source-based analysis for data-oriented gadget classification. First, Figure 4.1 shows an example of equivalent gadget classification across compilers and between source and binary analyses. The figure presents three instruction traces that correspond to the same conditional-type gadget. The binary-based instruction traces use VEX-IR as the intermediate language. The source-based instruction trace uses LLVM-IR. Aside from using different global addresses and temporary variable names, the binary-based gadgets are syntactically and semantically equivalent across compilers. The binary and source-based instruction traces are semantically but not syntactically equivalent.

Table 4.4 compares the classification results for this gadget across all test cases. The results show that classification is equivalent for this gadget. Note that although the binary instruction traces in Figure 4.1 do no explicitly reference `foreground` and `ppgrp`, the binary program's `data` and `bss` sections in the disassembly map the addresses to the names of these global variables.

	Binary, GCC (DOGGIE)	Binary, Clang (DOGGIE)	Source, Clang (LLVM)
Semantics	Conditional (=)	Conditional (=)	Conditional (=)
Address Parameters	`foreground` (Global)	`foreground` (Global)	`foreground` (Global)
Data Parameters	`ppgrp` (Global)	`ppgrp` (Global)	`ppgrp` (Global)

Table 4.4: Comparison of results for conditional gadget in "sudo".

Figure 4.2 is an example of a gadget that is not classified equivalently across

```
# Binary; compiled with GCC
t17 = LDle:I32(0x0805bb30)
t3 = GET:I32(offset=8)
t45 = CmpEQ32(t3,t17)
t44 = 1Uto32(t45)
t24 = t44
t46 = 32to1(t24)
t19 = t46
t47 = 1Uto8(t19)
t18 = t47
t48 = 8Uto32(t18)
t25 = t48
STle(0x0805bb20) = t25
```

```
# Binary; compiled with Clang
t2 = GET:I32(offset=8)
t1 = LDle:I32(0x08058b18)
t47 = CmpEQ32(t2,t1)
t46 = 1Uto32(t47)
t20 = t46
t48 = 32to1(t20)
t15 = t48
t49 = 1Uto8(t15)
t14 = t49
t50 = 8Uto32(t14)
t21 = t50
STle(0x08058b14) = t21
```

```
# Source; compiled with Clang
%63 = load i32* @ppgrp
%64 = icmp eq i32 %62, %63
%65 = zext i1 %64 to i32
store i32 %65, i32* @foreground
```

Figure 4.1: Conditional gadgets in "sudo".

platforms—showcasing some of the limitations of binary-based analysis. The considered gadget is an arithmetic-type gadget found in the `perform_io` function of "sudo." Indeed, in this example the Clang-compiled binary does not find the same Store instruction that corresponds to this gadget in any part of the function body—nor does it find any viable Store instructions to classify any gadget within the whole function. Thus, Figure 4.2 only shows the GCC-compiled binary and source-based instruction traces.

Table 4.5 shows that both classification methods choose the same type, but the parameters differ in scope. Based on the forward-pass scope inference technique presented in Section 3.3, Doggie determines that the parameters are local and distinct. However, the LLVM pass by Hu et al. (2016) uses the provided source code information to correctly identify the parameters as the same global data structure `io_bufs`. This semantic information is not available in the binary.

Additionally, the GCC-compiled binary version of "sudo" finds an extra dereferenced assignment gadget in the `perform_io` function that the source-based analysis does not find. Figure 4.3 presents the instruction trace for this gadget.

```
# Binary; compiled with GCC
t9 = GET:I32(offset=28)
t8 = Add32(t9,0xffffffec)
t10 = LDle:I32(t8)
t11 = Add32(t10,0x00000004)
t13 = LDle:I32(t11)
t15 = Add32(t9,0xfffffff4)
t3 = LDle:I32(t15)
t2 = Add32(t13,t3)
t17 = Add32(t9,0xffffffec)
t19 = LDle:I32(t17)
t20 = Add32(t19,0x00000004)
STle(t20) = t2
```

```
# Source; compiled with Clang
%iob.028 = load %struct.io_buffer** @iobufs
%iob.030 = phi %struct.io_buffer* [ %iob.0, %.loopexit ],
           [ %iob.028, %.lr.ph.preheader ]
%13 = getelementptr inbounds %struct.io_buffer* %iob.030,
      i32 0, i32 1
%48 = load i32* %13
%49 = add nsw i32 %48, %.lcssa
store i32 %49, i32* %13
```

Figure 4.2: Dereferenced arithmetic gadgets in the `perform_io` function in "sudo".

	Binary, GCC (DOGGIE)	Binary, Clang (DOGGIE)	Source, Clang (LLVM)
Semantics	Dereference, Arithmetic (+)	-	Dereference, Arithmetic (+)
Address Parameters	`*t19` (Local)	-	`iobufs` (Global)
Data Parameters	`t3` (Local), `*t11` (Local)	-	`iobufs` (Global)

Table 4.5: Comparison of results for arithmetic gadget in the `perform_io` function in "sudo."

4.4.2 unzip

The classification results for "unzip" highlight differences between compilers. Figure 4.4 presents an example of an arithmetic gadget across all three evaluation platforms. The Clang-compiled instruction traces express the semantics of the gadget in fewer instructions than the GCC-compiled version. The first instruction loads a global variable, the following instruction adds one to the variable, and the final instruction stores the new value back into the same global address.

Instead of directly referencing the global address, the GCC-compiled instruction trace references the data as a function parameter. It accesses the pointer to the data through an offset to the stack pointer register (`esp + 0x84`). Table 4.6 presents the

```
# Binary; compiled with GCC
t0 = GET:I32(offset=8)
t13 = LDle:I32(t0)
t15 = GET:I32(offset=28)
t14 = Add32(t15,0x00000010)
t16 = LDle:I32(t14)
t17 = Add32(t16,0x00000004)
STle(t17) = t13
```

Figure 4.3: Extra Dereference-Assignment gadget in `perform_io` function in GCC-compiled "sudo".

```
# Binary; compiled with GCC
t15 = GET:I32(offset=24)
t14 = Add32(t15,0x00000084)
t16 = LDle:I32(t14)
t5 = LDle:I32(t16)
t3 = Add32(t5,0x00000001)
STle(t16) = t3
```

```
# Binary; compiled with Clang
t4 = LDle:I32(0x0813c298)
t5 = Add32(t4,0x00000001)
STle(0x0813c298) = t5
```

```
# Source; compiled with Clang
%86 = load i32* getelementptr inbounds
      (%struct.Globals* @G, i32 0, i32 0, i32 7)
%91 = add nsw i32 %86, 1
store i32 %91, i32* getelementptr inbounds
      (%struct.Globals* @G, i32 0, i32 0, i32 7)
```

Figure 4.4: Arithmetic gadgets in "unzip".

classification results for this example. The primary difference is the scope and identity of the parameters between the compilers. Through purely static analysis, DOGGIE cannot infer that the value loaded in `t16` is the global variable `G`.

	Binary, GCC (DOGGIE)	Binary, Clang (DOGGIE)	Source, Clang (LLVM)
Semantics	Arithmetic (+)	Arithmetic (+)	Arithmetic (+)
Address Parameters	`t16` (Function Param.)	`G` (Global)	`G` (Global)
Data Parameters	`t16` (Function Param.)	`G` (Global)	`G` (Global)

Table 4.6: Comparison of results for arithmetic gadget in the `uz_opts` function in "unzip."

4.4.3 nginx

The reachability results for "nginx" are suitable for building a data-oriented programming exploit given the chosen vulnerability and the greater number of reach-

able gadgets compared to the other programs in the evaluation. The vulnerability (CVE 2013-2028) occurs when "nginx" processes a chunked transfer-encoded HTTP request. When parsing a large-enough chunked request, it is possible to trigger an integer signedness error and overflow a buffer on the stack.

Figure 4.5 shows the two data-oriented gadgets that exploit the signedness error. Both gadgets simulate dereferenced assignment operations. To demonstrate their semantic relation, Listing 4.1 presents the corresponding source code. This is part of the function `ngx_http_discard_request_body_filter` which "nginx" calls if the HTTP request is "chunked." This in turn calls `ngx_http_parse_chunked` on line 2, which contains the integer signedness vulnerability. The assignment in line 6 contains both gadgets from Figure 4.5. Thus, these gadgets are reachable from the vulnerable function and controllable from the dispatcher in line 1.

```
# Gadget 1
t9 = GET:I32(offset=32)
t8 = Add32(t9,0x0000001c)
t10 = LDle:I32(t8)
t11 = Add32(t10,0x00000010)
t13 = LDle:I32(t11)
t18 = GET:I32(offset=36)
t17 = Add32(t18,0x000000e0)
STle(t17) = t13
```

```
# Gadget 2
t9 = GET:I32(offset=32)
t8 = Add32(t9,0x0000001c)
t10 = LDle:I32(t8)
t14 = Add32(t10,0x0000000c)
t16 = LDle:I32(t14)
t18 = GET:I32(offset=36)
t20 = Add32(t18,0x000000dc)
STle(t20) = t16
```

Figure 4.5: Two dereferenced assignment gadgets in "nginx."

```
1 for ( ;; ) {
2     rc = ngx_http_parse_chunked(r, b, rb->chunked);
3 ...
4     if (rc == NGX_AGAIN) {
5         /* Two dereferenced assignment gadgets */
6         r->headers_in.content_length_n = rb->chunked->length;
7         break;
8     }
9 ...
```

Listing 4.1: Vulnerable code snippet in the function `ngx_http_discard_request_body_filter` in "nginx."

Table 4.7 presents the properties of the gadget parameters. Address Assembly

and Data Assembly reference the assembly instructions that build the parameters—normally a register plus an offset, composed by dereferencing. Address Source and Data Source reference the variable names in the source code from Listing 4.1. Note that `r->headers_in.content_length_n` and `rb->chunked->length` are of type `off_t`. Because "nginx" is compiled with `D_FILE_OFFSET_BITS=64`, the compiler forces variables of type `off_t` to be 64-bits in size. Thus, the resulting code in the 32-bit binary splits the assignment for `r->headers_in.content_length_n` between two data-oriented gadgets, each handling the data in four-byte chunks. It follows that both gadgets deal with semantically identical parameters. The difference being that the offsets differ by four bytes between the Address Assembly and Data Assembly rows, respectively.

	Gadget 1	Gadget 2
Address Parameter	`t17`	`t20`
Address Scope	Function Parameter	Function Parameter
Address Assembly	`edi + 0xe0`	`edi + 0xdc`
Address Source	`r->headers_in.content_length_n`	`r->headers_in.content_length_n`
Data Parameter	`t13`	`t16`
Data Scope	Function Parameter	Function Parameter
Data Assembly	`[[esi + 0x1c] + 0x10]`	`[[esi + 0x1c] + 0xc]`
Data Source	`rb->chunked->length`	`rb->chunked->length`

Table 4.7: Parameter details for the data-oriented gadgets from Figure 4.5 found in "nginx."

A data-oriented programming exploit for "nginx" uses these two dereferenced assignment gadgets as follows. First, an attacker sends a chunked HTTP request to a server running "nginx." The request is large enough that it fills the 1024 bytes of the first read and sets `rc` to `NGX_AGAIN` (line 4 of Listing 4.1). This also sets `rb->chunked->length` to a large number. Then, the data-oriented gadgets execute in line 6. Because the destination of the store is a signed type, `off_t`, `r->headers_in.content_length_n` becomes negative from the large value in `rb->chunked->length`. Next, `ngx_http_parse_chunked` executes a second time and the attacker sends over 4096 bytes, overflowing a vulnerable buffer on the stack. This

sets up the attacker to write arbitrary data to the stack and execute shellcode or even launch a return-oriented programming attack (as described in Vu (2013)).

4.5 Discussion

DOGGIE classifies data-oriented gadgets using program analysis and verification techniques. Evaluating how DOGGIE classifies gadgets compared to source-based analysis shows that it is viable for verifying short, foundational gadgets capable of delivering data-oriented programming exploits. Still, due to the lack of semantic information in binary-based analysis techniques DOGGIE does not classify the same gadgets as source-based analysis. Additionally, it cannot classify complex data-oriented gadgets. Even within the scope of binary-based analysis, gadget classification differs between compilers. The following addresses classification differences between compilers and the implications of omitting complex gadget classification.

4.5.1 Classification between Compilers

Gadget classification results differ between compilers due to how the compilers emit code. This affects the type and frequency of gadgets found in binaries. For instance, the "unzip" example in Section 4.4.2 shows one of the larger differences in classification results between Clang and GCC. The source code for the `uz_opts` function, which handles command line parameters, makes 30 modifications (some conditional) to a global data structure `uO` that stores unzip options. This is evident in the Clang classification results with a set of global gadgets in the `uz_opts` function. However, the GCC-compiled version is optimized in such a way that these operations do not form valid data-oriented gadgets.

This discrepancy points towards a systematic difference in how the compilers emit code and how the resulting code can be used in data-oriented gadgets. Recall that

there are three components to every data-oriented gadget: (1) parameter loading; (2) the simulated micro-operation (the "body" of the gadget); and (3) the final Store instruction. The following explores how differences in compilation can affect these three components.

Because every data-oriented gadget ends with a Store instruction, their presence and frequency is crucial for gadget discovery. Table 4.8 presents the total number of Store instructions for each program compiled under GCC and Clang. Δ is the relative difference between the counts for each compiler. "imlib2" and "unzip" show the largest relative difference between Store instruction counts. From Table 4.1, "unzip" indeed has 77.2% more total gadgets compiled under GCC than Clang. This is also confirmed in the mean "gadgets per function" and "gadgets per dispatcher" statistics in Table 4.2. However, this does not correspond to higher gadget frequency for a particular compiler. "imlib2", having 28.8% more store instructions under Clang than GCC, has 24% fewer total gadgets under Clang. Hence, the frequency of Store instructions does not consistently influence overall classification results between compilers.

	curl	imlib2	libtiff	nginx	optipng	sudo	unzip
GCC	6564	20555	19967	36813	12706	5450	11451
Clang	6090	28869	20434	38524	12548	5142	9755
Δ	7.2%	28.8%	2.2%	4.4%	1.2%	5.9%	14.8%

Table 4.8: Comparison of the number of Store instructions in each binary program compiled under GCC and Clang.

The gadget body is the sequence of instructions that determines the semantics of the micro-operation a gadget simulates. The length of a gadget body influences its complexity. The gadget statistics in Table 4.2 show that GCC on average emits code with longer gadgets than Clang by 16.4%. Due to the limitation on Doggie classifying complex gadgets, this may account for some misses in the GCC-compiled programs.

The remaining aspect considers how gadgets load parameters. The compiler influences each gadget by the code generation schemes it uses to load parameters from memory. Table 4.9 presents statistics on parameter-loading patterns for each program compiled under GCC and Clang. The three schemes it considers are (1) loading from a register (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`); (2) loading from the stack (through `ebp` or `esp`); and (3) loading directly from memory (constant address).

Application	Compiler	Registers	Stack	Constant
curl	GCC	2519	485	0
	Clang	1525	1207	16
imlib2	GCC	9005	8319	0
	Clang	8216	8742	0
libtiff	GCC	4203	2797	0
	Clang	2993	2940	0
nginx	GCC	5176	3559	93
	Clang	4904	3629	93
optipng	GCC	3558	1560	0
	Clang	2236	1076	34
sudo	GCC	1208	1266	102
	Clang	948	271	62
unzip	GCC	2348	1373	0
	Clang	828	497	367

Table 4.9: Comparison of parameter-loading strategies for gadgets in each program compiled under GCC and Clang.

Overall, both compilers prefer loading from registers. Aside from this, there is no general pattern distinguishing the two compilers. The differences are individual to each program. The previously explored "unzip" example highlights the gap between global variable classification between Clang and GCC. For Clang, the gadgets use a total of 367 constant-addressed parameters—whereas GCC reports zero. In these cases, these constant parameters are globally addressed and thus account for the large discrepancy between global gadgets. In general, an increase in constant-addressed parameters leads to an increase in globally classified gadgets.

How a compiler emits code affects the three components of a data-oriented gadget, thus influencing classification. However, what is not known are the rules and

optimizations that differ between compilers that affect the kinds of gadgets available. Ultimately, this means that the same gadgets are not guaranteed to be available for the same program across different compilers. Furthermore, this analysis shows that a data-oriented programming exploit developed for a binary under one compiler is not guaranteed to work under a different one. For defense, it also means that the degree of vulnerability—that is, how expressive of a DOP attack can be crafted—depends on how the program is compiled.

4.5.2 Simple versus Complex Gadgets

To understand the implications of omitting complex gadget identification in DOGGIE, Table 4.10 compares the number of verified gadgets versus the number of "potential" complex gadgets in each application. "Potential Gadget Total" includes all instruction sequences considered for classification. DOGGIE identifies potential complex gadgets by counting the number of types of arithmetic, logical, and conditional operations in a gadget's program slice. This approximates the actual number of complex gadgets in a program as it would require verification to know the true count.

Application	Compiler	Verified Gadget Count	Potential Complex Gadget Count	Potential Gadget Total	Verified Gadget Proportion	Complex Gadget Proportion
curl	GCC	1276	14	1298	98%	1%
	Clang	1174	13	1197	98%	1%
imlib2	GCC	3377	1516	5048	67%	30%
	Clang	2567	1976	4717	54%	42%
libtiff	GCC	2067	410	2572	80%	16%
	Clang	1403	479	1985	71%	24%
nginx	GCC	3284	285	3697	89%	8%
	Clang	3198	259	3615	88%	7%
optipng	GCC	1188	288	1582	75%	18%
	Clang	883	193	1125	78%	17%
sudo	GCC	988	49	1056	94%	5%
	Clang	456	22	486	94%	5%
unzip	GCC	1072	228	1350	79%	17%
	Clang	605	100	722	84%	14%

Table 4.10: Comparison of verified data-oriented gadget totals and potential complex gadgets that are omitted by DOGGIE.

The results in Table 4.10 show that the chosen programs have a majority of simple gadgets over complex ones. Applications like "curl" and "sudo" are made up

of over 90% simple gadgets. There is also not a large difference (less than 10 points) between gadget proportions between compilers. The one exception being "imlib2" which has the largest proportion of potential complex gadgets at 42%. This may suggest that libraries have more complex gadgets in general. For "sudo" and "unzip," there is a large difference (46%–53%) in gadget totals between compilers, but the proportion of complex gadgets remains low. Thus, based on the chosen applications in this evaluation, DOGGIE remains useful for data-oriented gadget classification in most cases. Some types of applications, such as libraries, may exhibit more complex gadgets than simple. However, even "imlib2" and "libtiff" contain multiple instances of assignment, dereference, and arithmetic gadgets (from Table 4.1) which is more than sufficient for crafting Turing-complete attacks (Hu et al. 2016).

The methodology presented in this work supports implementing complex data-oriented gadget verification; the process for computing the weakest precondition is the same. Extending DOGGIE to support complex gadget classification is a matter of identifying all of the relevant parameters and generating complex postconditions. These complex postconditions are simple predicates composed together (such as MOVE, LOAD, ARITHMETIC, etc. from Table 3.3). The difficulty comes in identifying which parameters fit to which variables in the postcondition predicate. There is no general method to do this. Trying all possible combinations leads to exponential computational complexity. A heuristic solution may analyze the operations used in a gadget and assign variables to operation types based on their usage (similar to the forward-pass analysis in Section 3.3).

Chapter 5

Conclusions

This thesis presents a methodology for classifying data-oriented gadgets in binary programs without source code. This is in contrast to current techniques that rely on source-based analysis. However, gadget classification without source code introduces difficulties due to the missing semantic information. To overcome this, this methodology uses a combination of data-flow and binary program analysis techniques for identification and formal methods for verification. Formal methods provide a guarantee of validity about the classification results.

Doggie (**D**ata-**O**riented **G**ad**g**et **I**dentifi**er**) is the prototype implementation of this classification methodology. This is a tool written in Python that, given a binary program, classifies data-oriented gadgets and determines their reachability with respect to a vulnerable function trace. Through the evaluation of a suite of programs, Doggie successfully classifies short, data-oriented gadgets capable of building data-oriented programming attacks. Comparing the classification results of Doggie to a source-based analysis—an LLVM pass by Hu et al. (2016)—shows some differences in gadget discovery. This is due to binary-based analysis missing semantic information like variable typing and pointers. Information like this helps to accurately identify gadget parameters. In binary-based analysis, this information is either partially re-

covered, approximated, or lost.

Furthermore, classification results also differ between programs under different compilers. This is due to how the compiler emits code. GCC and Clang use different conventions for loading function parameters and accessing global data which affects how gadget parameters are identified. In turn, this impacts the type and frequency of gadgets discovered in the same program compiled under different compilers. Thus, for practical data-oriented gadget analysis, the compiler must be considered. In this sense, the source-based analysis by Hu et al. (2016) is limited by the fact it uses LLVM for classification which limits analysis to Clang-compiled software. Exploits crafted using this information are not guaranteed to work on binaries under different compilers. From a security standpoint, this also hampers the awareness of how vulnerable a program is to data-oriented programming attacks. The source-based method does not consider the different compiler conventions and optimizations that ultimately affect the kinds of gadgets discoverable in the final binary.

DOGGIE, on the other hand, supports classification for software under any compiler. However, this comes at the cost of classification accuracy (due to the loss of semantic information). Despite this, the formally verified binary-based data-oriented gadget classification methodology and prototype implementation expand the scope of programs that can be analyzed for this class of exploit—including "common-off-the-shelf" binaries, closed-source binaries, and legacy programs. Over the previous source-based LLVM pass, DOGGIE provides gadget classification and reachability results that reflect the kinds of gadgets available by considering how the program was compiled.

Accurately classifying data-oriented gadgets in software is critical for assessing security vulnerabilities against data-only attacks. With this methodology and software prototype, security analysts can assess any generic binary for data-oriented gadgets and determine if a vulnerable function can trigger them. Furthermore, because the

methodology uses formal verification it provides a degree of guarantee about gadget properties and their reachability. As defenses against control-flow hijacking attacks become more widespread, data-only exploits become more viable attack vectors. In response, this research presents a solution that analyzes any generic binary for the building blocks of data-oriented programming attacks.

5.1 Related Work

Previous research has explored using program verification techniques to classify Return-oriented gadgets. Schwartz et al. (2011) developed a return-oriented programming exploit compiler that takes as input a binary program and an "exploit" program (written in a language similar to MinDOP in Table 3.1). It then uses program verification techniques to find suitable gadgets and compiler techniques to stitch them together and output a payload.

Previous work also explores defenses against data-oriented programming attacks. Data-flow integrity is a technique that instruments a program to protect data pointers from being corrupted (Castro et al. 2006). The instrumentation enforces the inherent data-flow of the program through pointer analysis—similar to how control-flow integrity forces a program to adhere to its static control-flow graph (Abadi et al. 2005). This is a general protection against data-only attacks and has not been tested specifically against DOP attacks. One of the drawbacks of this general technique is that it is computationally expensive to track all relevant data pointers in a program (incurs between 43%–104% overhead).

Specific defenses for DOP focus on embedded architectures and employ hardware assistance to reduce overhead. "HardScope" is hardware-assisted run-time scope enforcement for the RISC-V architecture (Nyman et al. 2017). This methodology mitigates DOP attacks by enforcing the lexical scope of variables at runtime. Another

defense called Operation Execution Integrity targets ARM-based embedded platforms (Sun et al. 2018). Operation Execution Integrity is an attestation method to mitigate both data-only and control-flow attacks. This security property verifies the control-flow and data integrity of a program at the operation level. Operation versus whole-program scope reduces overhead. Data integrity is also limited to "critical variables" which are automatically detected or manually identified. This also reduces overhead.

5.2 Future Work

Improvements to this methodology and implementation include complex data-oriented gadget verification. This work focuses on verifying gadgets with basic semantics. This was a practical decision as it is difficult to stitch together complex gadgets as they may have many side effects. However, for the complete analysis of gadgets in general binaries this work can be extended to handle gadgets with an arbitrary number of parameters and semantics.

To support complex gadget classification, DOGGIE should incorporate improved type and pointer recovery for gadget parameters. As shown in the case studies in Section 4.4, DOGGIE sometimes fails at correctly identifying gadget parameters. Advanced variable recovery techniques for binary programs (such as Balakrishnan & Reps (2010) and Lee et al. (2011)) may help DOGGIE more accurately identify gadgets parameters which in turn improves classification.

Other future work includes automating DOP exploit generation. After classifying gadgets and determining their reachability with respect to a vulnerable function, an attacker tests the "stitchability" of gadgets. Stitchability determines if the execution of one gadget flows into the execution of a subsequent gadget. So far, this is a manual process discovered through repeated execution of the program with combinations

of gadget sequences. An automated solution for testing stitchability can include a combination of symbolic and dynamic execution to test if two or more data-oriented gadgets can be executed in sequence.

Additionally, the verification methodology can be used as the basis of a formalism for modeling data-oriented programming attacks in general. Such a model can be studied to identify the necessary and sufficient properties of a general binary program to be vulnerable to this class of exploit.

Bibliography

Abadi, M., Budiu, M., Erlingsson, U. & Ligatti, J. (2005), Control-flow integrity, *in* 'Proceedings of the 12th ACM Conference on Computer and Communications Security', CCS '05, ACM, New York, NY, USA, pp. 340–353.
URL: *http://doi.acm.org/10.1145/1102120.1102165*

Aho, A., Lam, M., Sethi, R. & Ullman, J. (2006), *Compilers: Principles, Techniques, and Tools*, 2 edn, Pearson/Addison Wesley.

Andersen, S. & Abella, V. (2004), 'Data execution prevention. Changes to functionality in Microsoft Windows XP service pack 2, part 3: Memory protection technologies.'. Viewed 31 March 2017.
URL: *https://technet.microsoft.com/en-us/library/bb457155.aspx*

Balakrishnan, G. & Reps, T. (2010), 'WYSINWYX: What you see is not what you execute', *ACM Trans. Program. Lang. Syst.* **32**(6), 23:1–23:84.
URL: *http://doi.acm.org/10.1145/1749608.1749612*

Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D. & Boneh, D. (2014), Hacking blind, *in* 'Proceedings of the 2014 IEEE Symposium on Security and Privacy', SP '14, IEEE Computer Society, Washington, DC, USA, pp. 227–242.
URL: *http://dx.doi.org/10.1109/SP.2014.22*

Bletsch, T., Jiang, X., Freeh, V. W. & Liang, Z. (2011), Jump-oriented programming: A new class of code-reuse attack, *in* 'Proceedings of the 6th ACM Symposium on

Information, Computer and Communications Security', ASIACCS '11, ACM, New York, NY, USA, pp. 30–40.
URL: *http://doi.acm.org/10.1145/1966913.1966919*

Bosman, E. & Bos, H. (2014), Framing signals - a return to portable shellcode, *in* '2014 IEEE Symposium on Security and Privacy', pp. 243–258.

Brumley, D., Wang, H., Jha, S. & Song, D. (2007), Creating vulnerability signatures using weakest preconditions, *in* 'Proceedings of the 20th IEEE Computer Security Foundations Symposium', CSF '07, IEEE Computer Society, Washington, DC, USA, pp. 311–325.
URL: *https://doi.org/10.1109/CSF.2007.17*

Carlini, N. & Wagner, D. (2014), Rop is still dangerous: Breaking modern defenses, *in* 'Proceedings of the 23rd USENIX Conference on Security Symposium', SEC'14, USENIX Association, Berkeley, CA, USA, pp. 385–399.
URL: *http://dl.acm.org/citation.cfm?id=2671225.2671250*

Castro, M., Costa, M. & Harris, T. (2006), Securing software by enforcing data-flow integrity, *in* 'Proceedings of the 7th Symposium on Operating Systems Design and Implementation', OSDI '06, USENIX Association, Berkeley, CA, USA, pp. 147–160.
URL: *http://dl.acm.org/citation.cfm?id=1298455.1298470*

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H. & Winandy, M. (2010), Return-oriented programming without returns, *in* 'Proceedings of the 17th ACM Conference on Computer and Communications Security', CCS '10, ACM, New York, NY, USA, pp. 559–572.
URL: *http://doi.acm.org/10.1145/1866307.1866370*

Chen, S., Xu, J., Sezer, E. C., Gauriar, P. & Iyer, R. K. (2005), Non-control-data

attacks are realistic threats, *in* 'Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14', SSYM'05, USENIX Association, Berkeley, CA, USA, pp. 1–15.
URL: *http://dl.acm.org/citation.cfm?id=1251398.1251410*

Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P. & Zhang, Q. (1998), StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks, *in* 'Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7', SSYM'98, USENIX Association, Berkeley, CA, USA, pp. 1–15.
URL: *http://dl.acm.org/citation.cfm?id=1267549.1267554*

CVE (2018), 'CVE: Common vulnerabilities and exposures'. Viewed 16 April 2018.
URL: *https://cve.mitre.org*

De Moura, L. & Bjørner, N. (2008), 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', TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, pp. 337–340.
URL: *http://dl.acm.org/citation.cfm?id=1792734.1792766*

Dijkstra, E. W. (1976), *A Discipline of Programming*, 1st edn, Prentice Hall PTR, Englewood Cliffs, NJ, USA.

Flanagan, C. & Saxe, J. B. (2001), Avoiding exponential explosion: Generating compact verification conditions, *in* 'Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', POPL '01, ACM, New York, NY, USA, pp. 193–205.
URL: *http://doi.acm.org/10.1145/360204.360220*

Göktas, E., Athanasopoulos, E., Bos, H. & Portokalidis, G. (2014), Out of control:

Overcoming control-flow integrity, *in* 'Proceedings of the 2014 IEEE Symposium on Security and Privacy', SP '14, IEEE Computer Society, Washington, DC, USA, pp. 575–589.
URL: *http://dx.doi.org/10.1109/SP.2014.43*

Hu, H., Chua, Z. L., Adrian, S., Saxena, P. & Liang, Z. (2015), Automatic generation of data-oriented exploits, *in* '24th USENIX Security Symposium (USENIX Security 15)', USENIX Association, Washington, D.C., pp. 177–192.
URL: *https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/hu*

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

Kiriansky, V., Bruening, D. & Amarasinghe, S. P. (2002), Secure execution via program shepherding, *in* 'Proceedings of the 11th USENIX Security Symposium', USENIX Association, Berkeley, CA, USA, pp. 191–206.
URL: *http://dl.acm.org/citation.cfm?id=647253.720293*

Lattner, C. & Adve, V. (2004), Llvm: A compilation framework for lifelong program analysis & transformation, *in* 'Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization', CGO '04, IEEE Computer Society, Washington, DC, USA, pp. 75–.
URL: *http://dl.acm.org/citation.cfm?id=977395.977673*

Lee, J., Avgerinos, T. & Brumley, D. (2011), TIE: principled reverse engineering of types in binary programs, *in* 'Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th

February 2011'.

URL: *http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_3.pdf*

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. & Hazelwood, K. (2005), Pin: Building customized program analysis tools with dynamic instrumentation, *in* 'Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation', PLDI '05, ACM, New York, NY, USA, pp. 190–200.

URL: *http://doi.acm.org/10.1145/1065010.1065034*

Nyman, T., Dessouky, G., Zeitouni, S., Lehikoinen, A., Paverd, A., Asokan, N. & Sadeghi, A.-R. (2017), 'Hardscope: Thwarting DOP with hardware-assisted run-time scope enforcement', *CoRR* **abs/1705.10295**.

URL: *https://arxiv.org/abs/1705.10295*

Pax Team (2003*a*), 'Address space layout randomization (aslr)'. Viewed 31 March 2017.

URL: *https://pax.grsecurity.net/docs/aslr.txt*

Pax Team (2003*b*), 'PaX non-executable pages design and implementation'. Viewed 31 March 2017.

URL: *https://pax.grsecurity.net/docs/noexec.txt*

Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A. R. & Holz, T. (2015), Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications, *in* '2015 IEEE Symposium on Security and Privacy', pp. 745–762.

Schwartz, E. J., Avgerinos, T. & Brumley, D. (2011), Q: Exploit hardening made easy, *in* 'Proceedings of the 20th USENIX Conference on Security', SEC'11, USENIX

Association, Berkeley, CA, USA.

URL: *http://dl.acm.org/citation.cfm?id=2028067.2028092*

Shacham, H. (2007), The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), *in* 'Proceedings of the 14th ACM Conference on Computer and Communications Security', CCS '07, ACM, New York, NY, USA, pp. 552–561.

URL: *http://doi.acm.org/10.1145/1315245.1315313*

Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C. & Vigna, G. (2015), Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware, *in* 'Proceedings of the 2015 Network and Distributed System Security Symposium'.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C. & Vigna, G. (2016), SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis, *in* 'IEEE Symposium on Security and Privacy'.

Sun, Z., Feng, B., Lu, L. & Jha, S. (2018), 'OEI: operation execution integrity for embedded devices', *CoRR* **abs/1802.03462**.

URL: *http://arxiv.org/abs/1802.03462*

US-CERT (2014), 'OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160)'. Viewed 17 February 2017.

URL: *https://www.us-cert.gov/ncas/alerts/TA14-098A*

Vu, D. H. (2013), 'Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028)'. Viewed 1 June 2018.

URL: *https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html*

Weiser, M. (1981), Program slicing, *in* 'Proceedings of the 5th international conference on Software engineering', IEEE Press, pp. 439–449.

Appendix A

Source code for Algorithms and Formalisms

A.1 Source code for Listing 3.1, SetRelevantVariables()

The implementation in Listing A.1 differs from the pseudocode to explicitly handle when a VEX-IR statement defines a temporary variable versus a register. Lines 4 and 10 demarcate these two cases. Because of this, the object `relevant` maps a statement ID to a pair that denotes type (either temporary variable or register offset) and the associated value (temporary variable integer identifier or register offset number).

```python
def setRelevantVariables(irsb, relevant, targetId):
    for stmtId, stmt in enumerate(irsb.statements):
        if stmtId < targetId:
            if isinstance(stmt, pyvex.IRStmt.WrTmp):
                # If stmt defines relevant variable of proceeding stmt
                if ('tmp', stmt.tmp) in relevant[stmtId + 1]:
                    relevant[stmtId].add(('tmp', stmt.tmp))
                    # check if it is a Get reg expr
                    if isinstance(stmt.data, pyvex.expr.Get):
                        # add offset to relevant set
                        relevant[stmtId].add(('offset', stmt.data.offset))
                    for t in getTmpsFromRHS(stmt):
                        relevant[stmtId].add(('tmp', t))
                # If stmt does not define relevant variable of proceeding
                    stmt
                else:
                    # ...then add that variable to proceeding stmt's
                        relevant variables
                    relevant[stmtId + 1].add(('tmp', stmt.tmp))
            # Same procedure for Put stmt's but checking offsets
            elif isinstance(stmt, pyvex.IRStmt.Put):
                if ('offset', stmt.offset) in relevant[stmtId + 1]:
                    relevant[stmtId].add(('offset', stmt.offset))
                    if isinstance(stmt.data, pyvex.expr.Get):
                        relevant[stmtId].add(('offset', stmt.data.offset))
                    for t in getTmpsFromRHS(stmt):
                        relevant[stmtId].add(('tmp', t))
                else:
                    relevant[stmtId + 1].add(('offset', stmt.offset))
    return relevant
```

Listing A.1: Python source code for pseudocode in Listing 3.1.

A.2 Source code for Listing 3.2, BackwardProgramSlice()

The implementation in Listing A.2 differs from the pseudocode to explicitly handle when a VEX-IR statement defines a temporary variable versus a register. Lines 7 and 10 demarcate these two cases. Because of this, the object `relevant` maps a statement ID to a pair that denotes type (either temporary variable or register offset) and the associated value (temporary variable integer identifier or register offset number).

```
1  def makeBackwardSlice(irsb, targetId):
2      relevant = initRelevantVariables(irsb, targetId)
3      relevant = setRelevantVariables(irsb, relevant, targetId)
4      pslice = []
5      for stmtId, stmt in enumerate(irsb.statements):
6          if stmtId < targetId:
7              if isinstance(stmt, pyvex.IRStmt.WrTmp):
8                  if ('tmp', stmt.tmp) in relevant[stmtId + 1]:
9                      pslice.append(stmtId)
10             elif isinstance(stmt, pyvex.IRStmt.Put):
11                 if ('offset', stmt.offset) in relevant[stmtId + 1]:
12                     pslice.append(stmtId)
13     return pslice
```

Listing A.2: Python source code for pseudocode in Listing 3.2.

A.3 Source code for Listing 3.3, BDFA()

The implementation in Listing A.3 differs from the pseudocode to explicitly handle when a VEX-IR statement defines a temporary variable versus a register. Lines 5 and 23 demarcate these two cases.

```
def BDFA(v, bslice, istack, statements):
    if not bslice:
        return
    i = bslice.pop()
    if isinstance(statements[i], pyvex.IRStmt.WrTmp):
        if v[0] == 'tmp':
            if statements[i].tmp == v[1]:
                subseq = [i]
                if isinstance(statements[i].data, pyvex.expr.Get):
                    substack = []
                    BDFA(('offset', statements[i].data.offset), bslice,
                        substack, statements)
                    subseq += substack
                else:
                    rhs = getTmpsFromRHS(statements[i])
                    for t in rhs:
                        substack = []
                        rbslice = list(bslice)
                        BDFA(('tmp', t), rbslice, substack, statements)
                        subseq += substack
                istack += subseq
            else:
                BDFA(v, bslice, istack, statements)
    elif isinstance(statements[i], pyvex.IRStmt.Put):
        if v[0] == 'offset':
            if statements[i].offset == v[1]:
                rhs = getTmpsFromRHS(statements[i])
                subseq = [i]
                for t in rhs:
                    substack = []
                    BDFA(('tmp', t), bslice, substack, statements)
                    subseq += substack
                istack += subseq
            else:
                BDFA(v, bslice, istack, statements)
```

Listing A.3: Python source code for pseudocode in Listing 3.3.

A.4 Source code for Listing 3.4, GetGadget()

The implementation in Listing A.4 repeats logics for the "address" and "data" parameters of the target Store instruction. Function `isInteresting()` checks if an instruction stack contains at least one Load statement (i.e., a viable gadget).

```
def getGadget(stmt_idx, stmt, irsb):
    bslice = makeBackwardSlice(irsb, stmt_idx)

    # Dictionary of lists to hold interesting instruction stacks
    stackd = defaultdict(list)

    astack = [] # Instruction stack from stmt.addr.tmp
    tmpslice = list(bslice)
    try:
        BDFA(('tmp', stmt.addr.tmp), tmpslice, astack, irsb.statements)
        # Removing duplicates from instruction stack
        # Don't care about preserving order since it is always descending
            anyways
        astackSet = set(astack)
        astack = list(astackSet)
        astack.sort(reverse=True)
    except AttributeError as e:
        pass

    dstack = [] # Instruction stack from stmt.data.tmp
    tmpslice = list(bslice)
    try:
        BDFA(('tmp', stmt.data.tmp), tmpslice, dstack, irsb.statements)
        # Removing duplicates from instruction stack
        # Don't care about preserving order since it is always descending
            anyways
        dstackSet = set(dstack)
        dstack = list(dstackSet)
        dstack.sort(reverse=True)
    except AttributeError as e:
        pass

    if (isinstance(stmt.addr, pyvex.expr.Const) or (isInteresting(astack,
        irsb))
            and isInteresting(dstack, irsb)):
        stackd['addr'] = astack
        stackd['data'] = dstack

    return stackd
```

Listing A.4: Python source code for pseudocode in Listing 3.4.

A.5 Source code for Listing 3.6, GetPotentialGadgets()

The implementation in Listing A.5 uses built-in functions from the angr binary program analysis framework to recover functions and loops from a binary and traverse those data structures.

```
# For each function in the program (determined by CFG)
for f in cfg.kb.functions.values():

    # Find loops in this function
    loops = proj.analyses.LoopFinder(functions=[f])

    # For each loop in this function
    for loop in loops.loops:

        isInterestingLoop = False
        # For each basic block in the function
        for n in loop.body_nodes:

            # Grab the VEX-IR super block from this address
            irsb = proj.factory.block(n.addr).vex

            for stmt_idx, stmt in enumerate(irsb.statements):
                # Looking for STOREs
                if isinstance(stmt, pyvex.IRStmt.Store):
                    instr_stack = getGadget(stmt_idx, stmt, irsb)
                    potGadgets.append(Gadget(set(), set(),
                        angr.analyses.code_location.CodeLocation(n.addr,
                        stmt_idx), instr_stack, f.addr, loop))

            # Also check for CALLS in the loop and follow them through
            followSuccessors([], n.addr, f.addr, cfg, potGadgets, loop)
```

Listing A.5: Python source code for pseudocode in Listing 3.6.

A.6 Source code for computing Weakest Precondition of a GCL-like program

```
def WP(prog, Q, M, R, t, stmts, proj):
    while prog:
        cmd = stmts[prog.pop()]
        if isinstance(cmd, pyvex.IRStmt.Store):
            data = cmd.data.tmp
            addr = None
            if isinstance(cmd.addr, pyvex.expr.RdTmp):
                addr = t[cmd.addr.tmp]
            elif (isinstance(cmd.addr, pyvex.expr.Const) and
                isGlobalVariable(cmd.addr.con.value,
                proj.loader.main_object.sections_map)):
                addr = BitVecVal(cmd.addr.con.value, 32)
            Q = substitute(Q, (M[addr], t[data]))
        elif isinstance(cmd, pyvex.IRStmt.WrTmp):
            if isinstance(cmd.data, pyvex.expr.Load):
                src = None
                if isinstance(cmd.data.addr, pyvex.expr.RdTmp):
                    src = t[cmd.data.addr.tmp]
                elif (isinstance(cmd.data.addr, pyvex.expr.Const) and
                    isGlobalVariable(cmd.data.addr.con.value,
                    proj.loader.main_object.sections_map)):
                    src = BitVecVal(cmd.data.addr.con.value, 32)
                Q = substitute(Q, (t[cmd.tmp], M[src]))
            elif isinstance(cmd.data, pyvex.expr.Binop):
                op1, op2 = cmd.data.args
                binop = getOperatorType(cmd.data.op)
                if comparisonOperator(cmd.data.op):
                    Q1 = Implies(binop(op1, op2), substitute(Q, (t[cmd.tmp],
                        BitVecVal(1, 32))))
                    Q2 = Implies(Not(binop(op1, op2)), substitute(Q,
                        (t[cmd.tmp], BitVecVal(0, 32))))
                    Q = And(Q1, Q2)
                else:
                    Q = substitute(Q, (t[cmd.tmp], binop(op1, op2)))
            elif isinstance(cmd.data, pyvex.expr.Get):
                Q = substitute(Q, (t[cmd.tmp], R[cmd.data.offset]))
            elif isinstance(cmd.data, pyvex.expr.RdTmp):
                Q = substitute(Q, (t[cmd.tmp], t[cmd.data.tmp]))
        elif isinstance(cmd, pyvex.IRStmt.Put) and isinstance(cmd.data,
            pyvex.expr.RdTmp):
            Q = substitute(Q, (R[cmd.offset], t[cmd.data.tmp]))
    return Q
```

Listing A.6: Python implementation for computing weakest precondition according to the semantics in Figure 3.2.