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

Zachary D. Sisco
Wright State University
sisco.8@wright.edu

Adam R. Bryant
Wright State University
adam.bryant@wright.edu

ABSTRACT

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 bypass 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.

KEYWORDS

Data-oriented exploits, binary program analysis, data flow analysis, program verification, formal methods

ACM Reference format:
Zachary D. Sisco and Adam R. Bryant. 2019. Verifying Data-Oriented Gadgets in Binary Programs to Build Data-Only Exploits. In *Proceedings of , , (2019),* 13 pages.
DOI:

1 INTRODUCTION

"Data-only" attacks are a class of exploits that take advantage of memory corruption vulnerabilities to manipulate a program's data plane. Instead of hijacking control flow by manipulating return addresses and function pointers, these attacks cause harm by changing how a program executes its logic and decision-making routines through the manipulation of data values [7]. Data-only attacks respect a program's inherent control flow, so they are difficult to mitigate using control-flow hijacking protections like Address Space Layout Randomization, Data Execution Prevention, and Control-flow Integrity.

Data-oriented programming (DOP) is a type of data-only code-reuse attack that stitches together sequences of data manipulation instructions to simulate computation [13]. In DOP, attackers chain together sequences of instructions called *data-oriented gadgets* that simulate common micro-operations—such as assignment, arithmetic, and conditionals–to craft expressive, even Turing-complete, exploits [13]. Correctly classifying data-oriented gadgets and their properties is critical for detecting DOP attacks and automating data-oriented exploit generation.

This paper presents a methodology (Section 3) for classifying data-oriented gadgets in binary programs without the availability of compiler instrumentation or source code. The classification methodology uses data-flow analysis and verification conditions to identify and verify data-oriented gadgets and their properties. The prototype developed in this research demonstrates the feasibility of crafting DOP exploits in binaries without source code. Current state-of-the-art techniques to discover data-oriented gadgets rely on the availability of source code; however, by classifying data-oriented gadgets without source code, the methodology and resulting prototype presented in this paper 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 determine first, if a binary has existing data-oriented gadgets, and second, if those gadgets can be triggered from a given vulnerability. The evaluation (Section 4) of the classification prototype also explores the differences in classifying data-oriented gadgets with and without source code. Additionally, it shows how compilers introduce differences in the kinds of gadgets available in a binary and how they are discovered.

2 BACKGROUND

Memory corruption vulnerabilities are errors such as stack and heap buffer overflows, integer overflows, use-after-free, double-free, and format string vulnerabilities [7]. Control-flow exploits take advantage of these vulnerabilities to 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 [22]). Control data are most often pointers that attackers insert to redirect the program's program counter to execute code at a particular address provided by the attacker.

Defense mechanisms to mitigate control flow attacks focus on protecting a program's *control data.* Stack canaries [8] prevent control data from being overwritten by inserting randomized "canary" values in between local variables and control data on the stack. Memory policies, such as "W⊕X" [20] and Data Execution Prevention [3], prevent user-supplied data from being executed.

Address Space Layout Randomization [19] randomizes the locations the operating system loads program sections—such as the stack, heap, and libraries—to mitigate code reuse attacks like return-oriented programming [22]. In attacks that depend on knowing

*Research funded by Edaptive Computing Inc. through Air Force Contract FA8650-14-D-1724-0003.

the addresses of libraries and program sections, randomizing these addresses makes it harder for the attacks to succeed. Program shepherding [14] and Control-flow Integrity [1] are methods that ensure a program follows its control-flow graph during execution, thus thwarting attacks that hijack control from a program.

Control-oriented defenses do not mitigate *data-only* attacks [7]—also called non-control data attacks, or data-oriented attacks. *Data-only* attacks differ from control-flow attacks in that they do not alter the program's control flow structures like branches, branch targets, function pointers, or return addresses. Instead, data-only attacks tamper with data that affects which logic or decision-making routines are executed [7], such as configuration data, user identity data, user input data, passwords, keys, randomized values, and system call parameters. Manipulating non-control data causes harm through sensitive information leakage, privilege escalation, and arbitrary code execution.

Listing 1 shows one example of a data-only attack in *wu-ftpd* (version 2.6.0), a free FTP server daemon. 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 [7]. Then, line 7 temporarily escalates to root privileges in order to invoke `setsockopt()` [7]. 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 [7].

```
1  struct passwd { uid_t pw_uid; ... } *pw;
2  ...
3  int uid = getuid();
4  pw->pw_uid = uid;
5  printf(...); // format string vulnerability
6  ...
7  seteuid(0); // set root id
8  setsockopt(...);
9  ...
10 seteuid(pw->pw_uid); // set unprivileged user id
11 ...
```

Listing 1: Vulnerable code snippet in `wu-ftpd`.

2.1 Data-Oriented Programming

Data-oriented programming (DOP) is a general method for constructing data-only attacks [13]. Given a vulnerable program, DOP builds Turing-complete data-only attacks capable of a high degree of expressiveness and arbitrary computation [13]. The methodology resembles return-oriented programming [22] whereas DOP 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 from short sequences of instructions in a vulnerable program. These differ from code gadgets in return-oriented programming in that data-oriented gadgets must execute without deviating from the program's established control flow structures [13]. Additionally, data-oriented gadgets can only persist the output of their operations to memory—whereas code gadgets in return-oriented programming can use memory or registers [13]. 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.

The parts of a data-oriented exploit are the data-oriented gadgets, the gadget dispatcher, and the memory error. The *gadget dispatcher* chains and sequences a series of data-oriented gadgets to form an attack [13]. Attackers control the selection and activation of gadgets by writing data via the program's memory error [13].

Like gadgets, the gadget dispatcher 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 at each iteration. 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.

The evaluation of DOP by [13] shows that data-oriented gadgets are as prevalent in software as return-oriented gadgets and that it is possible to construct Turing-complete exploits that bypass current memory protections.

2.2 Data-Oriented Programming Without Source Code

This paper explores the feasibility of constructing DOP 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 automatically constructing DOP exploits and detecting them. 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 "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 binary and if they can be triggered from a given vulnerability.

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 [13], 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. [13] defines a basic language to express data-oriented gadgets, MinDOP (Table 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.

Table 1: MinDOP language by [13], and how it relates to C instructions

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.		

For example, Figure 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 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 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. We do this using the *angr* binary program analysis framework [24], which uses static and concolic analysis techniques. Angr uses VEX-IR, an assembly-like language with single static assignment (SSA), which it exposes via Python bindings [23]. This approach simplifies data-flow analysis because the language abstracts away irrelevant differences between instruction set architectures while explicitly enumerating instruction side-effects.

To find gadgets that can be continually invoked from a gadget dispatcher, we start with those that start from loops. We use angr's built-in control flow scanning to extract loops from each function of the binary, following function calls invoked in the loops and then scanning for gadgets at their function call targets. We mark these gadgets as *reachable* from the original enfolding loop.

Data-oriented gadgets always end with a store instruction, so to identify the gadgets, we consider each store instruction and analyze its antecedent statements within the same basic block. A store instruction has two arguments (the source register and the pointer to the destination memory address), so we trace backwards from it to collect statements that process the data that flows to each argument. This tracing is done through backward static program slicing. Given a program P, a backward slice at program point p, with set of variables V representing the statement arguments, contains only those preceding statements in P that affect the variables in V at p [29]. Thus the backward slice starts at a store instruction and returns the antecedent statements that affect the source value and destination address.

Gadget identification at the basic-block level reduces the complexity of program slicing because there are no loops or conditionals in a basic block. Listing 2 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 an 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 [2]. The difference here is that Listing 2 returns the path (sequences of instructions) that the target variable is live at.

```
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 2: Pseudocode for backward data-flow analysis algorithm. Note that i.LHS and i.RHS refer to the left and right-hand sides of the statement i.

We call Listing 2 which recursively calls itself for each argument to the store instruction. The resulting pair of instruction sequences is a potential data-oriented gadget.

The following example in Listing 3 demonstrates how backward static program slicing followed by the backward data-flow analysis algorithm produces two program slices that trace the definitions of the arguments to a store instruction. In the example, the variables `t37` and `t36` represent the store instruction's destination address and source data, 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 (line 13) loads the value and stores the result in `t33`. Then, the final instruction (line 14) 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`.

```
1   # Original basic block
2   t11 = GET:I32(offset=28) # 28 = EBP
3   t31 = Add32(t11,0xffffef70)
4   t33 = LDle:I32(t31)
5   t34 = Add32(t11,0xffffefec)
6   t36 = LDle:I32(t34)
7   t37 = Add32(t33,0x00000010)
8   STle(t37) = t36
9
10  # Program slice tracing t37
11  t11 = GET:I32(offset=28)
12  t31 = Add32(t11,0xffffef70)
13  t33 = LDle:I32(t31)
14  t37 = Add32(t33,0x00000010)
15
16  # Program slice tracing t36
17  t11 = GET:I32(offset=28)
18  t34 = Add32(t11,0xffffefec)
19  t36 = LDle:I32(t34)
```

Listing 3: Backward static program slice example in VEX-IR.

The pseudocode in Listing 4 wraps all the algorithms together for whole-program 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. The "getGadget()" function creates a program slice given a store instruction and basic block, and returns a pair of instruction traces computed from Listing 2. 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 4 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.

Unlike return-oriented programming gadgets—which are single sequences ending in a return instruction—data-oriented gadgets have two sequences of instructions to consider, one for each of the store instruction's two arguments. The identification algorithms presented here (Listings 2, 4) describe how to build program slices

```
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 4: This pseudocode presents an algorithm for whole-program identification of potential data-oriented gadgets.

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

[13] presents an approach to classify 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 methodology follows the work of [21] 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* [11].

Thus, gadget classification becomes a problem of deriving the weakest precondition of program slices. Characterized by [11], *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. [12] adapted GCL and predicate transformers to derive verification conditions for Java programs. [5] adapted these rules again for use in binary analysis, which is also the application for 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 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 [12]. 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 designed for binary analysis.

Table 2: A GCL-like syntax for specifying programs to apply predicate transformers to derive weakest preconditions

GCL *Statement*	s	::=	$x := e$
		|	**assume** e
		|	$s\,;\,s$
		|	$s \square s$
VEX-IR *Expression*	e	::=	t
		|	r
		|	m
		|	c
		|	$t \diamond t$
		|	$t \diamond c$
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	$\in$	$\mathbb{Z}$

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 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 from VEX-IR to the GCL-like syntax is trivial. Table 3 presents an example for translating an addition-type gadget.

Table 3: Translating from VEX-IR to GCL for an addition-type gadget

VEX-IR	Lifts to	GCL
`t2 = Load(0x880123)`	$\Longrightarrow$	$\mathcal{T}[2] := \mathcal{M}$[0x880123];
`t1 = Add(t2, t3)`	$\Longrightarrow$	$\mathcal{T}[1] := \mathcal{T}[2] + \mathcal{T}[3]$;
`Store(t3) = t1`	$\Longrightarrow$	$\mathcal{M}[\mathcal{T}[3]] := \mathcal{T}[1]$

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)` then lifts to:

(**assume** $(\mathcal{T}[2] \diamond_c \mathcal{T}[3])$; $\mathcal{T}[1] := 1$;) $\square$

(**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 2. These are the predicate transformers as adapted by [5] for binary program analysis. The Python source code implementation takes as input a sequence of statements in GCL syntax and postcondition, and derives the weakest precondition according to the rules in Figure 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 2: Semantics for deriving weakest precondition of a GCL-like program (adapted from [5]).

3.2.1 Example Arithmetic Gadget Classification. Listing 5 presents an example gadget performing an `ADD` operation with a constant. The VEX-IR statements in Listing 5 lift to the following GCL-like statements (abbreviated to $s_1; s_2; s_3$) in Equation 1.

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

Listing 5: VEX-IR example program slice demonstrating an arithmetic `ADD` gadget.

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

For this to be a valid arithmetic gadget, we derive a valid weakest precondition for Equation 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 shows the application of the rules in Figure 2 on the gadget in Equation 1. The resulting weakest precondition is a reflexive equality and is trivially valid, hence showing that the gadget is indeed a valid arithmetic gadget.

To classify gadgets of different types, we specify the postconditions presented in Table 4. $\diamond_a$ 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

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

Figure 3: Application of weakest precondition derivation rules for example gadget in Equation 1. The initial value of Q is $(\mathcal{M}[\texttt{0x805c7e8}] = \mathcal{T}[5] + 2)$.

to the store instruction—the destination and value, respectively. With the exception of Conditional operations, these postconditions resemble the MinDOP syntax presented in Table 1.

Table 4: Postconditions for verifying data-oriented gadget semantics

Name	Parameters	Postcondition
Assign	**Out**, **In**	**Out** = **In**
Deref (Load)	**Out**, **In**	**Out** = $\mathcal{M}$[**In**]
Deref (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	$((x \diamond_c y) \Rightarrow \textbf{Out} = 1) \wedge (\neg(x \diamond_c y) \Rightarrow \textbf{Out} = 0)$

3.3 Scope Inference and Optimizations for Classifying Gadgets

Note that pointer information for the inputs is not included in Table 4. Since DOP 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, depending on the 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 whether the variable is scoped as a global variable, function parameter, or local variable. 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 scope of 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 whether the variable is 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 program in Listing 6 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 4) 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 6: 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 [10]. 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 whole-program analysis algorithm in Listing 4.

3.5 Identifying Gadget Dispatchers

Similar to [13] 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 [13] 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 section. 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.

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 [24]. The tool's name is Doggie—**D**ata-**O**riented **G**ad**g**et **I**dentifi**e**r. Doggie identifies and verifies the semantics of data-oriented gadgets in binary programs. The tool leverages the SMT solver Z3 [10] for verification.

Additionally, the tool determines the *reachability* of gadgets to vulnerable functions in a binary program. This process first leverages Intel's Pin [17], 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. Doggie does not verify gadgets that exhibit *complex* semantics. We define "complex" as a gadget that has more than two movement operations (assignment or dereference) and more than one binary operation. Although this implementation decision omits certain 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. We discuss the implications of this decision in Section 4.5.

4.1 Evaluation

To evaluate how accurately Doggie classifies data-oriented gadgets in binary programs we compare the classification results of the tool with [13]'s source-based gadget discovery tool. [13]'s gadget discovery tool uses LLVM version 3.5.0 [15]. 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 selected programs include: *curl*, *imlib2*, *libtiff*, *nginx*, *optipng*, *sudo*, and *unzip*.

In addition to reporting classification results for data-oriented gadgets, the evaluation 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 [9]. Because the source-based tool from [13] does not report gadget reachability, we only provide reachability results of the binary programs using Doggie.

4.2 Classification Results

Table 5 presents the results of data-oriented gadget classification for binary and source-based programs using Doggie and [13]'s LLVM pass, respectively. For gadget scopes, 'G' is Global, 'H' is Hybrid (mixed between global and local), and 'L' is Local. The table classifies gadgets according to two dimensions—semantics and scope. Semantics are the type of micro-operations that the gadget simulates. Scope defines the context of the parameters for the gadget. 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 [13].

Table 5 shows there are relatively few conditional gadgets, and more arithmetic, dereferencing, and assignment gadgets. Additionally, the data in Table 5 demonstrates inconsistency in classifying 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 both semantics and scope for the three cases.

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. The use of

Table 5: Data-oriented gadget identification results from Doggie across binaries compiled with GCC version 4.9.2 and Clang version 3.5.0. †Indicates source-based results due to [13]

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

Table 6: Data-oriented gadget reachability results w.r.t vulnerabilities from the CVE database [9]

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 7.41.0	2015-3144	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 1.4.7	2016-3994	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 2.5.6	2017-9935	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 1.4.0	2013-2028	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 0.7.6	2016-3982	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 1.8.3p1	2012-0809	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 6.0	2015-7696	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

verification conditions to check the semantics ensures that these gadgets are not false positives.

Furthermore, there is as much of a difference between compilers as there is between binary and source-based analysis. This is an area for further research and Section 4.4 begins to explore these differences through case studies. However, a complete analysis of the differences between compilers for classification is out of scope for this paper.

4.3 Reachability Results

Table 6 presents the results of data-oriented gadget reachability given a vulnerable function trace for binary programs using Doggie. For each program we trigger a disclosed vulnerability to produce a vulnerable function trace.

Again, the reachability results in Table 6 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. However, 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 [13] 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.

4.4 Case Studies

Analyzing programs with our binary analysis method and the LLVM pass presented in [13] produced results that seemed inconsistent. In some cases, our binary method detects many more gadgets, but in others it fails to classify some gadgets caught by the LLVM pass. There are also differences in the results depending on which

compiler was used to create the binary. We use results from three programs—a systems utility (*sudo*), a file decompression tool (*unzip*), and an HTTP server (*nginx*)—to explore the nature of these differences.

4.4.1 Sudo. The analysis of *sudo* exemplifies the differences in these results. The program contains an arithmetic-type gadget in its `perform_io` function that each of the methods classify differently.

Table 7 shows that both binary- and source-based classification methods identified the gadget as the same type, but they found the scope of its parameters to be different. Based on the forward-pass scope inference technique presented in Section 3.3, Doggie determines the parameters are local and distinct; however, the LLVM pass by [13] uses the richer semantic information from the source code to correctly identify the parameters as the same global data structure `io_bufs`. This semantic information is not available in the binary. When the binary is compiled with Clang, Doggie does not find the gadget's corresponding store instruction–or any viable store instruction in the function–for this gadget to use so it is not classified.

Table 7: Evaluation results for an arithmetic gadget in *sudo*

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

4.4.2 Unzip. The results for *unzip* highlight differences between the code emitted by compilers. The *unzip* program contains a gadget in the `uz_opts` function that all three modalities discovered (Figure 4 lists the instructions).

The Clang-compiled instruction trace (shown in the middle box) express the gadget's semantics in fewer instructions than the GCC-compiled version (shown in the top box). 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.

```
# 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: Arithmetic gadget in `uz_opts` function in *unzip*.

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`). This results in differences in gadget classification in the scope and identity of the parameters between the compilers (see Table 8). Since it relies on purely static analysis, Doggie cannot infer that the value loaded in `t16` is the global variable `G`.

Table 8: Evaluation results for an arithmetic gadget in *unzip*

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

4.4.3 Nginx. The reachability results for *nginx* are suitable for building a DOP exploit given the chosen vulnerability and the greater number of reachable gadgets compared to the other programs in this 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.

Listing 7 shows two data-oriented gadgets that exploit the signedness error. Both gadgets simulate dereferenced assignment operations. 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. These gadgets are reachable from the vulnerable function and controllable from the dispatcher in line 1.

```
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 7: Two gadgets in vulnerable code snippet from *nginx*.

Note that the variables `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 store for `r->headers_in.content_length_n` between two gadgets, each handling the data in four-byte chunks.

A DOP 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 2 of

Listing 7). This also sets `rb->chunked->length` to a large number. Then, the 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 [28].

4.5 Discussion

Doggie classifies data-oriented gadgets using binary program analysis techniques and verification conditions. Evaluating how Doggie classifies gadgets compared to source-based analysis shows that it is viable for verifying short, foundational gadgets capable of delivering DOP 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 for the reasons discussed in the following sub-sections.

4.5.1 Classification between Compilers. Gadget classification results differ between compilers due to how the compilers emit code. In some cases, gadgets found from code compiled with one binary are not found in another. This affects the type and frequency of gadgets Doggie identifies. For instance, the *unzip* example in Section 4.4.2 shows one of the larger differences in classification results between code compiled with 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 `u0` 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 whether the resulting code can be used in data-oriented gadgets. Recall that there are three components to every data-oriented gadget: (1) the store instruction, (2) the simulated micro-operation (the "body"), and (3) the parameters. Most of Doggie's classification differences arise from parameter inference, which is caused by differences in how each compiler loads those parameters.

The compiler influences each gadget by the code generation schemes it uses to load parameters from memory. Table 9 presents statistics on parameter-loading patterns for each program compiled under GCC and Clang. Parameters can be (1) loaded from a register (`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`); (2) loaded from the stack (through `ebp` or `esp`); and (3) loaded directly from memory (i.e., from a constant address).

Both GCC and Clang prefer loading from registers. Aside from this, we saw no discernible pattern to generalize code emitted by the two compilers—the differences are specific to each program. The previously explored *unzip* example highlights the differences in how Doggie classifies variables as globally scoped. For the Clang-compiled version, the gadgets use a total of 367 constant-addressed parameters—whereas in the GCC-compiled version, Doggie reports zero. In these cases, these constant parameters are globally addressed which leads to an increase in globally classified gadgets. This accounts for why Doggie reports such a large discrepancy between the two versions.

Table 9: Parameter-loading stats for GCC and Clang-compiled binaries

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

The way a compiler emits code also affects the resulting properties of data-oriented gadgets, thus influencing classification. This means the same gadgets are not guaranteed to be available for the same program across different compilers. Our analysis shows that a DOP exploit developed for a binary compiled with one compiler is not guaranteed to work for a binary compiled for a different one. This creates a possible avenue for defense in terms of diversifying the attack surface of a computer system. It also means the degree of vulnerability—that is, the measure of how expressive a DOP attack can be crafted—also depends on how the program is compiled. Future work should explore more deeply how the rules and optimizations used in various compilers affect the kinds of gadgets available.

4.5.2 Simple versus Complex Gadgets. Since we focused only on simple (rather than complex) gadget identification, we also explored the ramifications of this choice. 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. Table 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.

The results in Table 10 show that the chosen programs have a majority of simple gadgets over complex ones. We approximate that 90% of gadgets are simple in applications like *curl* and *sudo*. There is also not a large difference (less than 10 points) between gadget proportions between compilers—with the exception of *imlib2* which has the largest proportion of potential complex gadgets at 42%. 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,

Table 10: Verified data-oriented gadgets vs. potential complex gadgets (omitted by Doggie)

Application	Compiler	Verified Gadgets	Potential Complex Gadgets	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%

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 5) which is sufficient for crafting Turing-complete attacks [13].

Nevertheless, future work should extend Doggie to support complex gadget classification for completeness. To do so is a matter of identifying all the relevant parameters and generating more complex postconditions. The process for computing the weakest precondition is the same. These complex postconditions are combinations of simple predicates composed together (such as Assign, Load, Arithmetic, etc. from Table 4). The difficulty comes in identifying which parameters fit to which variables in the postcondition predicate. There is yet no general method to do this. Trying all possible combinations leads to exponential computational complexity. A heuristic solution could 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).

5 CONCLUSIONS

This paper 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. Gadget classification without source code introduces difficulties due to missing semantic information, but to overcome this, our methodology uses a combination of data-flow and binary program analysis techniques for identification, and uses 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**e**r) 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 DOP attacks. Comparing the classification results of Doggie to a source-based analysis shows some differences in gadget discovery. This is due to binary-based analysis missing semantic information like variable typing and pointers. In binary-based analysis, this information is either partially recovered, approximated, or lost.

Classification results also differ between programs under different compilers due to how the compilers emit code. GCC and Clang use different conventions for loading function parameters and accessing global data which affects how gadget parameters are identified. 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 same sense, the source-based analysis by [13] 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 DOP 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. Despite this, the formally verified binary-based data-oriented gadget classification methodology and 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 DOP attacks.

5.1 Related Work

Previous research has explored using program verification techniques to classify return-oriented gadgets. [21] 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). It then uses program verification techniques to find suitable gadgets and compiler techniques to stitch them together and output a payload. Other research has also studied modeling information flow in binary programs [26] and verifying binary program properties through formal semantics [25]. These techniques are related to the ones used in this work.

Previous work also explores defenses against data-only attacks. Data-flow integrity is a technique that instruments a program to protect data pointers from being corrupted [6]. 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 [1]. 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 [18]. 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 [27]. This is an attestation method that verifies the control-flow and data integrity of a program at the operation level. Data integrity is also limited to "critical variables" which are either automatically or manually identified. This also reduces overhead.

5.2 Future Work

Improvements to this work include complex data-oriented gadget verification (as described in Section 4.5.2). To support complex gadget classification, Doggie should also 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 [4] and [16]) 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 may include a combination of symbolic and dynamic execution to test if two or more data-oriented gadgets can be executed in sequence. Such a tool can help software analysts concretely assess DOP vulnerability, as the presence of gadgets is not sufficient to determine exploitability.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM, New York, NY, USA, 340–353. DOI: http://dx.doi.org/10.1145/1102120.1102165

[2] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2 ed.). Pearson/Addison Wesley.

[3] Starr Andersen and Vincent Abella. 2004. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. (2004). https://technet.microsoft.com/en-us/library/bb457155.aspx Viewed 31 March 2017.

[4] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages. DOI: http://dx.doi.org/10.1145/1749608.1749612

[5] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. 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, 311–325. DOI: http://dx.doi.org/10.1109/CSF.2007.17

[6] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 147–160. http://dl.acm.org/citation.cfm?id=1298455.1298470

[7] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 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, 1–15. http://dl.acm.org/citation.cfm?id=1251398.1251410

[8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 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, 1–15. http://dl.acm.org/citation.cfm?id=1267549.1267554

[9] CVE. 2018. CVE: Common Vulnerabilities and Exposures. (2018). https://cve.mitre.org Viewed 16 April 2018.

[10] Leonardo De Moura and Nikolaj Bjørner. 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, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[11] Edsger Wybe Dijkstra. 1976. *A Discipline of Programming* (1st ed.). Prentice Hall PTR, Englewood Cliffs, NJ, USA.

[12] Cormac Flanagan and James B. Saxe. 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, 193–205. DOI: http://dx.doi.org/10.1145/360204.360220

[13] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. 969–986. DOI: http://dx.doi.org/10.1109/SP.2016.62

[14] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 191–206. http://dl.acm.org/citation.cfm?id=647253.720293

[15] Chris Lattner and Vikram Adve. 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, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[16] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 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*. http://www.isoc.org/isoc/conferences/ndss/11/pdf/5_3.pdf

[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 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, 190–200. DOI: http://dx.doi.org/10.1145/1065010.1065034

[18] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N Asokan, and Ahmad-Reza Sadeghi. 2017. Hardscope: Thwarting DOP with hardware-assisted run-time scope enforcement. *CoRR* abs/1705.10295 (2017). https://arxiv.org/abs/1705.10295

[19] Pax Team. 2003. Address Space Layout Randomization (ASLR). (2003). https://pax.grsecurity.net/docs/aslr.txt Viewed 31 March 2017.

[20] Pax Team. 2003. PaX non-executable pages design and implementation. (2003). https://pax.grsecurity.net/docs/noexec.txt Viewed 31 March 2017.

[21] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Conference on Security (SEC'11)*. USENIX Association, Berkeley, CA, USA, 17. http://dl.acm.org/citation.cfm?id=2028067.2028092

[22] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 552–561. DOI: http://dx.doi.org/10.1145/1315245.1315313

[23] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium*.

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[25] Zachary D. Sisco and Adam R. Bryant. 2017. A semantics-based approach to concept assignment in assembly code. In *Proceedings of the 12th International Conference on Cyber Warfare and Security*. ACPIL, Reading, UK, 341–351.

[26] Zachary D. Sisco, Patrick P. Dudenhofer, and Adam R. Bryant. 2017. Modeling information flow for an autonomous agent to support reverse engineering work.

The Journal of Defense Modeling and Simulation 14, 3 (2017), 245–256. https://doi.org/10.1177/1548512916670784

[27] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. 2018. OEI: Operation Execution Integrity for Embedded Devices. *CoRR* abs/1802.03462 (2018). http://arxiv.org/abs/1802.03462

[28] Dang Hoang Vu. 2013. Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028). (2013). https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html Viewed 1 June 2018.

[29] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.