2019 IEEE European Symposium on Security and Privacy (EuroS&P)

STEROIDS for DOPed Applications: A Compiler for Automated Data-Oriented Programming

Jannik Pewny
Ruhr-Universität Bochum, Germany
jannik.pewny@rub.de

Philipp Koppe
Ruhr-Universität Bochum, Germany
philipp.koppe@rub.de

Thorsten Holz
Ruhr-Universität Bochum, Germany
thorsten.holz@rub.de

***Abstract*—The wide-spread adoption of system defenses such as the randomization of code, stack, and heap raises the bar for code-reuse attacks. Thus, attackers utilize a scripting engine in target programs like a web browser to prepare the code-reuse chain, e.g., relocate gadget addresses or perform a just-in-time gadget search. However, many types of programs do not provide such an execution context that an attacker can use. Recent advances in data-oriented programming (DOP) explored an orthogonal way to abuse memory corruption vulnerabilities and demonstrated that an attacker can achieve Turing-complete computations without modifying code pointers in applications. As of now, constructing DOP exploits requires a lot of manual work—for every combination of application and payload anew.**

In this paper, we present novel techniques to automate the process of generating DOP exploits. We implemented a compiler called STEROIDS that leverages these techniques and compiles our high-level language SLANG into low-level DOP data structures driving malicious computations at run time. This enables an attacker to specify her intent in an application- and vulnerability-independent manner to maximize reusability. We demonstrate the effectiveness of our techniques and prototype implementation by specifying four programs of varying complexity in SLANG that calculate the Levenshtein distance, traverse a pointer chain to steal a private key, relocate a ROP chain, and perform a JIT-ROP attack. STEROIDS compiles each of those programs to low-level DOP data structures targeted at five different applications including GStreamer, Wireshark, and ProFTPd, which have vastly different vulnerabilities and DOP instances. Ultimately, this shows that our compiler is versatile, can be used for both 32-bit and 64-bit applications, works across bug classes, and enables highly expressive attacks without conventional code-injection or code-reuse techniques in applications lacking a scripting engine.

***Index Terms*—Data-Oriented Programming, Exploitation, Compiler, Steroids, Slang**

I. INTRODUCTION

Attackers have to overcome more and more obstacles to exploit an application given that defense mechanisms such as stack protections, data-execution prevention (DEP), address space layout randomization (ASLR) and control-flow integrity (CFI) [1]–[3] are nowadays widely deployed. Modern defenses often require that an attacker adapts her exploit to the current state of the application she is attacking. For this reason, many modern exploits target browsers or PDF readers, which come with a built-in scripting engine. Utilizing the computational capabilities of these scripting engines, e. g., JavaScript or Flash, the attacker can repeatedly leverage a memory error, probe memory, find gadgets just-in-time, and perform complex computations to make the code-reuse chain compatible to the target application's security model. However, system defenses like execute-only memory (XOM) and its relatives [4]–[7], CPI [8] and Readactor [9], [10] aim to mitigate even advanced just-in-time code-reuse attacks.

An alternative line of attacks targets non-control data [11]. These so-called data-oriented attacks (DOAs) can have consequences just as severe as code-reuse attacks, but neither alter code pointers nor rely on the present code randomization. Despite DOAs being less well-explored than code-reuse attacks, there have been efforts to ensure data integrity [12]–[19]. Nevertheless, efficient mitigation of DOAs with strong security guarantees remains a challenging and open problem.

Data-oriented programming (DOP) is an extension of DOA, which manipulates non-control data to use the application's own operations to perform arbitrary computations of the attacker's choice. Thus, they could provide the execution context modern code-reuse methods need, even in applications which lack a scripting engine. We dub this a *bring your own scripting engine (BYOSE)* attack, which is more versatile and potentially even more harmful than DOAs alone, and arguably makes securing data flow an even more pressing topic.

The state of the art in DOP is mostly concerned with searching for DOP gadgets [20] or considers rather simple programs [21]. However, using DOP gadgets for non-trivial DOP programs is left to manual work, as the resulting exploits are highly application- and vulnerability-dependent. While existing work showed that diverse and powerful DOP gadgets are available, it is unclear which DOP gadgets are actually necessary or can be expressed through others, or which minimal sets of DOP gadgets achieve useful expressiveness in different DOP attack modes.

In this paper, we close this gap: we provide novel insights into the flexibility of DOP, introduce a high degree of automation for the construction of DOP exploits, and demonstrate that complex attack payloads work across different applications and vulnerabilities. More specifically, we present a high-level programming language called SLANG, which can be used to describe DOP programs. Using descriptions of the DOP gadgets available from a specific vulnerability of a specific application (conceptually, the output of the gadget search [20], [22]), we use an automated way to craft the data structures that trigger DOP gadgets to carry out the computation described in the DOP program. Since the SLANG scripts are designed to be application-independent, the available DOP gadgets

DOI 10.1109/EuroSP.2019.00018

usually do not immediately yield all necessary operations to express the attacker's intent. To tackle this problem, we support and encode *recipes* for various operations, i. e., ways to combine them to express either one another or more high-level operations. Implicitly, this creates a graph structure, which we call an *Operations Graph (Op-Graph)*. Given such an Op-Graph, we translate the attacker's script to use only DOP gadgets present in the target application. As such, most of the manual effort to create a DOP program for one application can be reused to exploit other vulnerabilities or applications with different sets of DOP gadgets.

To demonstrate the practicality of our techniques, we implemented our approach in a tool called STEROIDS. It can automatically build different DOP programs for different applications with different defense mechanisms. E. g., one of our scripts expects to be run on a randomized binary [23], and searches gadgets at runtime to ultimately mount a just-in-time code-reuse attack. This effectively bypasses the protection offered by, say, Binary Stirring [24] or Compiler-assisted Code Randomization [25]. One key element is that the attacker's script does not have to be modified to be compiled for different vulnerabilities or different applications. Furthermore, our experiments include examples to highlight various features of our compiler, namely (i) to *bootstrap* additional DOP gadgets, (ii) to compile to *branch-free* code to compensate for lacking conditional jumps, and (iii) to support an *interactive mode* if only a single DOP gadget can be executed at a time.

Ultimately, our results show that DOP is not esoteric, but can indeed be utilized to reliably execute complex attack payloads in target programs. These BYOSE attacks transfer the possibility of advanced code-reuse attacks, such as JIT-ROP, to target applications without a built-in scripting engine.

In summary, our main contributions are as follows:

- We provide novel insights regarding the minimal requirements for successful DOP exploitation. We can push the boundaries by bootstrapping new DOP instructions, utilizing branch-free code and an interactive mode.
- We develop novel techniques to automate the construction of DOP programs and enable the reuse of exploits across target applications and vulnerabilities.
- We present our prototype implementation STEROIDS, which can compile exploits specified in our high-level scripting language SLANG for a given target application.
- We demonstrate the practicality of our techniques and prototype by implementing complex programs in SLANG and automatically compiling them to DOP programs in binary form. Our SLANG scripts include ROP chain relocation and a runtime gadget search. We show that BYOSE enables just-in-time code-reuse attacks for applications *without* a built-in scripting engine and thus, extends the set of potential targets.

II. TECHNICAL BACKGROUND

Before diving into the details of our approach, we briefly introduce the necessary technical background information on data-oriented programming needed to understand the building blocks of our method.

Code-reuse attacks. The motivation for return-to-libc attacks [26], return-oriented programming (ROP) [27] and its variants [28]–[31] was the wide-spread adoption of the $W \oplus X$ policy, which prohibits the execution of writable data sections rendering code injection infeasible. These so called code-reuse attacks have in common that they reuse pieces of existing code, which are called ROP gadgets, and employ a mechanism to chain these gadgets by creating or modifying code pointers. Code-randomization defenses [24], [32]–[38] shuffle and modify gadgets, forcing attackers to relocate their ROP chain or even search new gadgets to construct a new ROP chain on the fly (JIT-ROP [39], [40]). To mitigate just-in-time code-reuse attacks, many defenses have been proposed that hide code and code pointers [7], [9], [10], [41]–[43] or aim to preserve the control flow [1]–[3], [8], [44].

Data-only attacks. Leveraging memory corruptions to modify non-control data is an orthogonal attack vector [11], [22], [45]. The attacker corrupts data structures or data pointers to modify the data flow. In this way, the program's logic can be tricked into leaking sensitive information like private keys [22] or passing attacker-controlled content to access control data structures or critical functions such as `execve`. The existing efforts to prevent DOAs [12]–[19] impose a high performance overhead or do not provide strong security guarantees.

Data-oriented programming. Similar to ROP, DOP aims to achieve a high degree of expressiveness, but without modifying code pointers. The attacker utilizes a memory corruption vulnerability to modify existing data structures or to inject new data structures. The ultimate goal of DOP is to repurpose the logic of existing pieces of code (DOP gadgets) to perform the attacker desired computation (DOP operations). However, arbitrarily chaining of gadgets spread across the program, as is possible with ROP, is not possible for DOP. A certain proximity in the control-flow graph is required to ensure continuous execution and selection of the next DOP gadget. Conceptually, two methods to chain DOP gadgets exist: First, the *non-interactive mode* utilizes an existing loop in the program with DOP gadgets in its loop body to ensure continuous execution. Additionally, a mechanism is necessary that selects the next data structure like a linked list or an array of structs. Often the loop condition needs to be modified to keep the loop running, which can be accomplished with the memory corruption or the execution of the first DOP gadget. Since the payload contains the trigger for the memory corruption and the data for all instructions, the attacker needs no further interaction with the target application. Second, in the *interactive mode*, the attacker initiates the loop iterations separately by repeatedly leveraging the memory corruption and feeding the data structures that trigger the desired DOP gadgets. The motivation for this mode is that the attacker may not be able to extract state of the target program, e.g., if the memory corruption has no read capabilities. In this case adapting a code-reuse payload on the attacker side is severely hampered. That being said, if the attacker can extract

state, she can *adapt* following DOP gadgets for an even more efficient *adaptive interactive mode*. For the remainder of the paper, we will assume that this is not the case, though.

We define the following three steps to launch a DOP attack:

Gadget search: Find DOP gadgets that are reachable directly or indirectly by the memory corruption vulnerability. The scope depends on the bug, but can range from structures on the active stack frame to the whole program. Furthermore, the outputs of DOP gadgets must be readable by the following DOP gadgets. This step also involves searching gadget dispatcher loops or establish an outside mechanism to chain gadgets.

DOP instance setup: Collect path constraints to reach the basic blocks containing DOP gadgets and setup data structure templates that trigger desired DOP gadgets. In the non-interactive mode, one also needs to setup the mechanism that feeds the required data structures and triggers the next cycle.

Payload preparation: The native gadgets of the DOP instance often provide constrained and unusual operations. For example, some DOP instances lack a controllable program counter, conditional operations, or even basic arithmetic and data movement. Thus, this step involves bootstrapping a convenient set of operations and creating a stream of operations that compensates lacking features. Finally, the collection of data structure contents and inputs for the memory corruption must be compiled.

Previous work [20] focused on the automation of the gadget search. While they also performed DOP instance setup and payload preparation, they did so in a mostly manual fashion. In contrast, Block-Oriented Programming [21] provides a higher degree of automation, but its program synthesis is NP-hard. Thus, it can mainly solve the constraints for DOP programs that are not too complex, and rather belongs to the step of DOP instance setup.

In this paper, we pursue orthogonal research by focusing on automating the payload preparation. We leave the gadget search to related work and interface the DOP instance setup. That is, we define a file format for this stage and process it accordingly in the payload preparation. The key design idea to avoid NP-hardness is to work with unconstrained data flow and inputs in later steps, i. e., the gadgets are arbitrarily stitchable and work for all inputs. Almost all manual work that is left to the attacker belongs to her custom payload or to the DOP instance setup, which can be seen as retargeting our prototype compiler STEROIDS to a new "platform".

III. AUTOMATED DOP EXPLOIT COMPILER

In this section, we describe the techniques that enable automated compilation of DOP programs under consideration of constrained DOP gadgets, and provide a high-level overview of STEROIDS's components and their interactions.

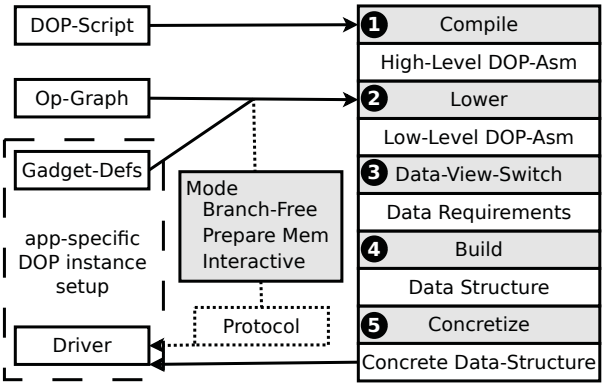

Fig. 1: Workflow of STEROIDS

A. Assumptions and Attacker Model

Our assumptions are aligned with previous research and we adopt the general scenario given in the previously reported DOP instances: First, we assume that the target application is protected with DEP, ASLR, and advanced control-flow hijack mitigations such as fine-grained CFI or fine-grained load-time code randomization. Second, we assume the presence of a memory corruption vulnerability. Our test cases make use of both stack-based vulnerabilities and heap-based vulnerabilities (see Section V). Lastly, we assume that an input triggering the vulnerability is known.

B. Overview

Figure 1 provides an overview of the inputs, stages, and intermediary artifacts of our approach. Note that we focus on the concepts first and give examples for the artifacts and more detail about their format later in this section.

The dashed rectangle on the bottom left represents the DOP instance setup with its two components: The *gadget definitions* and the *driver*. The gadget definitions specify the necessary constraints for the data structures driving the execution of a specific DOP gadget present in the application. Naturally, these constraints are highly specific to both the application and the vulnerability. Conceptually, they directly result from findings of the gadget search [20]–[22], but are set manually in our approach. However, to correctly judge the necessary effort, one has to keep in mind that a gadget definition in the five target applications we evaluate (see Section V) is on average only about ten lines long, and that one only needs to define about half a dozen of them (see Section VI-B). In context of compilers, the gadget definitions are analogous to the instructions of a new architecture.

The *driver* represents the interface to the vulnerable application (e. g., it writes to a file or opens a TCP connection to trigger the initial vulnerability). Given that this is highly specific to the exploit and the application's input stream, we again rely on the attacker to supply the *driver*.

Now follows the centerpiece of this paper, the payload preparation. We separate the automatic compilation of DOP

exploits into five stages, which are reflected in the design of our prototype implementation (see Figure 1). After the attacker specified her intent using our high-level programming language, SLANG, we first *compile* (❶) her DOP script into an assembly-like format: *High-Level DOP-Asm*. This High-Level DOP-Asm is independent of the application, that is, it may use DOP gadgets which the application does not provide.

In step ❷, we use a set of recipes, which encode how to express DOP gadgets through other DOP gadgets. We use these recipes and the Op-Graph they implicitly define through their interdependencies, to *lower* the High-Level DOP-Asm into Low-Level DOP-Asm. The latter uses only DOP gadgets from the gadget definitions, i. e., only DOP gadgets available for this specific vulnerability and application. For the most part, the recipes and Op-Graph can be reused for multiple applications, unlike the application-specific gadget definitions.

The Low-Level DOP-Asm is meant to be read with execution in mind, but we have to think in terms of data driving the execution in context of DOP. Thus, in the *data-view-switch* (❸), we use the content of the gadget definitions. This results in the *data requirements*, which are the constraints for a data structure that would execute the DOP script if it were placed in the vulnerable application.

In step ❹, we actually *build* the *data structure* fulfilling the formerly created *data requirements*. Since this involves potentially expensive constraint-solving, the *data structures* can contain placeholders instead of concrete values. An additional *concretization* (❺) step can then quickly replace constants or addresses leaked at runtime.

Lastly, we pass the final data structure back to the *driver*, in order to execute it in the target application.

C. Modes

The *mode* element in Figure 1 represents an accumulation of compiler flags triggering different program transformations, and indicates the DOP gadget supply mode required for the specific target DOP instance. There are three separate, but partially interlocking optional modes: branch-free, memory preparation, and interactive.

Branch-Free. If one cannot synthesize conditional jumps for the application, one cannot easily express if/else statements. Listing 1 shows how to use arithmetics[1] to express case distinction [46]: All operations are executed on the right side, whereas one needs to conditionally skip operations on the left side. That is, the right side can do without a `conditional goto`. This transformation of an operation must fulfill two properties: First, it must only have an effect, if the `here`-bit is set and second, it may only assume that the values it uses are properly initialized, if the `here`-bit is set.

Thus, we compute a `here`-bit at the start of every basic blocks by comparing a `state`-variable to a basic block's ID. Then, we can simply change the `state`-variable instead of using `goto`s. The result is that the execution sequence of basic blocks is no longer important, only that they are executed often enough.

[1]This example uses an equality operator, which produces a one if the values are equal and a zero otherwise. Alternatively, one can place alternative values in an array and compute different indices, or use conditional DOP gadgets.

Listing 1: Conditional and Branch-Free Code

```
t = x + y
if(z == 3):
    r = t
```

```
t = x + y
here = z == 3
r = here * t + (1-here) * r
```

Memory Preparation. When DOP gadgets are placed in separate buffers, e.g., in the interactive mode, it may not be possible to place persistent variables next to the DOP gadgets using them. However, one can use DOP gadgets to place the variables somewhere in memory, before the remainder of the DOP script is executed. This phase therefore collects all variables, translates their initial values into immediate values used by DOP gadgets, and transforms the script in order to use the addresses the variables are written to.

Interactive. A `goto` DOP gadget is not necessary in the interactive mode, because the attacker can decide which operation to send next. However, this also places the burden to decide which operation to send next on the attacker. This is inherently problematic if the execution state is not known to the attacker, e. g., if her DOP script requires case distinctions or loops. Luckily, due to the branch-free transformation, it only matters that an operation is executed often enough, because additional executions have no effect. Thus, one big loop would theoretically suffice to execute any program, e. g., if the DOP gadgets are saved in a linked list, which the attacker can corrupt to form a circle. However, executing only what is needed is obviously more performant.

For reducible programs [47], we can automatically generate a *protocol* which details how often to execute which sequence of basic blocks. Using well-known techniques, we analyze the CFG, inline functions, and dissect the program into the building blocks of structured programming: sequences, which are executed in-order, selections (if/else), for which both the true path and the false path are executed once, and iterations, for which we rely on the attacker for annotations in the DOP script to hint the number of repetitions.

D. Artifacts

Now that we have discussed the stages of our approach, let us give more details on the used artifacts.

High-Level Language: SLANG. Our tool STEROIDS provides a high-level language for the development of DOP programs. Our STEROIDS programming language, SLANG, provides

- compound expressions
- typed variables (addresses, bytes, int16, int32...)
- type-sensitive arrays
  - constant initialization: strings, int-arrays, hex-dumps
  - automatic length-variable for constant initialization
- structured control-flow
  - if/else blocks
  - loops: for, while, repeat ... until, infinite
  - break/continue
- (recursion-free) procedures

This is in contrast to Qool [48] for ROP or MinDOP [20] for DOP, which provide only "list-of-statements" languages. Since the exemplary scripts in Section IV show that our language is rather straightforward and conventional, we refrain from showing the formal grammar of SLANG in this paper, both for brevity and to omit technicalities.

The compilation from SLANG to High-Level DOP-Asm itself is also straightforward (e. g., the elements of structures control flow are compiled to checks, labels and `goto`s). Compound expressions are parsed and compiled into single two-operand operations for each node in the parse tree, using temporary variables, if necessary. Array accesses multiply the index with the size of an array element, before using the result as an offset after dereferencing the array's address.

DOP assembly. While SLANG is used to define the attacker's intent, DOP assembly is used as an intermediary step, but also as a convenient language for the recipes and for inline-use in SLANG. It is low-level and assembly-like, where each line holds a command with its operands. Additionally, it features

- typed variables (addresses, bytes, int16, int32...)
- conditional operations
- labels as jump-targets
- macros, e. g.,
  - to generate unique labels or variable names
  - to reserve only one temporary variable, if a gadget is used multiple times
  - to perform compile-time computation
- compiler directives, e. g., to tell the compiler
  - to advance a program counter
  - to increase the size of a packet

For a simple example, refer to Listing 2. Note that the computation could also be done byte-wise, such that one requires only $2^8 * 4$ instead of 2^{32} loop iterations.

Listing 2: Recipe to synthesize an `add`, using `mov, dec, inc` and a `conditional goto`. Suffixes for the bit-width of the operations and operands are omitted.

```
add dst src
    int cpy
    mov cpy src
    :start
        if_zero_goto cpy :end
        dec cpy
        inc dst
    goto :start
    :end
```

Operations Graph. Recipes encode how to synthesize higher-level operations from simpler ones. The Op-Graph is created simply by parsing a set of recipes and drawing edge sets from an operation to the operations used in a specific recipe.

To illustrate our approach, a recipe to express the `add` DOP operation (`*p += *q`) is given in Listing 2. The Op-Graph in Figure 2 includes the dependencies defined in the recipe, and shows that `add` uses five other DOP operations, indicated by the bold edges. Furthermore, it shows that the application only has two DOP gadgets: `load` (`*p = **q`) and a `conditional goto`, indicated by the bold nodes.

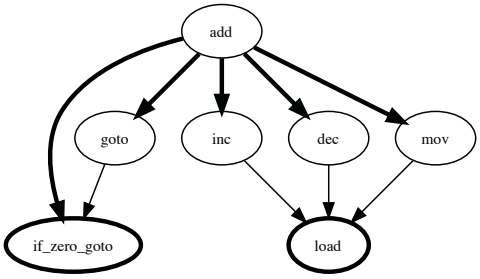

Fig. 2: Op-Graph corresponding to the recipe from Listing 2 for synthesizing an `add`.

A recursive graph search on the Op-Graph starting at the `add`-node, shows each DOP operation in this recipe requires only other DOP operations, which themselves require only DOP gadgets present in the application. Thus, this recipe can provide an `add` DOP operation to the attacker, even though the application does not have an `add` DOP gadget.

A `goto` can be synthesized using a `conditional goto` with an always-true dummy condition, and a `mov` (`*p = *q`) can be implemented using a `load` with one additional indirection, but expressing `inc/dec` through a `load` may be a little surprising: In Section V-B we describe how to use a lookup table to achieve this feature.

The lowering of High-Level DOP-Asm to Low-Level DOP-Asm requires only macro expansion along the found paths to the DOP gadgets: We successively substitute a DOP operation with the body of its recipe, taking care to substitute recipe's parameters with the DOP operation's arguments, until we end up using only DOP gadgets. These are then expanded using their gadget definitions to form the data requirements.

For convenience, if a DOP operation cannot be synthesized because of missing DOP gadgets, our compiler generates an And/Or-graph from the recipes, which reports all possible alternatives of which DOP operations one would have to implement to complete said recipes. Furthermore, we designed our Op-Graphs to be combinable trivially, so simple file concatenation allows reusing recipes for other applications.

DOP-Gadget Definition. In DOP, one uses skillfully crafted data structures to drive the program to perform specific operations. Thus, at one point, one has to switch from the execution-perspective to the data structure-perspective that is actually used in the application (see Listing 3 for an example). The language we use to define data structures is fairly simple: It supports symbols and constants, which may be initialized to specific addresses and values, memory dereferences and offsets, where the latter offers a typed array-index formulation as syntactic sugar.

Listing 3: Artificial example of a DOP gadget.

```
struct s { int *value, *from, *to; };
struct s *ptr = ...;
// memory corruption of ptr
if(ptr->value == 3)
    *(ptr->to) = *(ptr->from);
```

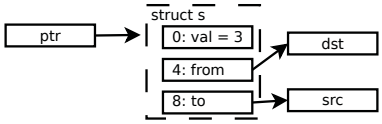

Fig. 3: Data structure to invoke Listing 3's DOP gadget.

Listing 4: Gadget definition / Data structure requirements for the data structure from Figure 3 to feed the DOP gadget from Listing 3. The address of `ptr` is an input to the compiler, while `src/dst` are the parameters of the DOP gadget.

```
mov dst src
    ptr deref offset(0) = 3
    ptr deref offset(4) = src
    ptr deref offset(8) = dst
```

For example, Listing 4 defines the data structure from Figure 3 to trigger a `mov` operation. These data requirements are then transformed into constraints for the Z3 SMT Solver: We require that, at a certain position in the buffer, there is an address (`ptr`). Somewhere else in the buffer, there must be the value `3`, and `ptr` must point to it. This value must be followed by another pointer (`from`), which must have the value given by the attacker as the second parameter (`src`), and yet another pointer (`to`), which must have the value of the first parameter (`dst`).

While it is not necessary for this simple example, our compiler can handle multiple dereferences and offsets on both sides of the equation, e. g., to write to a member in a struct.

Protocol. The protocol holds nested sequences of basic blocks and, if they occur in a loop, optionally how often to execute that loop. We opted for S-Expressions as an easy-to-parse format. E. g., we translate the protocol

$$(BB_1, (3, BB_2, BB_3), BB_4)$$

into the trace

$$BB_1, BB_2, BB_3, BB_2, BB_3, BB_2, BB_3, BB_4.$$

Note that a protocol is only required in the interactive mode, where DOP gadgets can be combined arbitrarily. Thus, the protocol is not part of the *data requirements* and only needs to be passed to the *driver*.

E. Implementation Details

We implemented STEROIDS on Linux in Python and use the Z3 SMT Solver to solve the constraints of the *data requirements*. We compiled DOP programs for 32-bit and 64-bit Linux target applications. Conceptually, it should also work for other operating systems since the generated data structures by themselves are not OS-dependent.

Optimizations. STEROIDS is meant to provide maximum flexibility in its compile targets, since essentially every DOP-instance offers different DOP gadgets, and we therefore chose generality over efficiency. Still, when emitting DOP assembly, STEROIDS tries to use the simplest DOP operation. E. g., by emitting `inc/dec` instead of `add/sub`, or by avoiding the oftentimes rather expensive `neq/eq` DOP gadgets when comparing against a constant, in favor of calculating the condition of a conditional DOP gadget at compile time.

Furthermore, we allow annotations for weighted edges in the Op-Graph. By default, a DOP gadget native to the target application has a weight of one and a `goto` a weight of ten to account for repeated execution of a loop-body. Our compiler can then sums the weights of all the used DOP gadgets and pick the recipes with the lowest cumulative weight. STEROIDS also reduces swapping when lowering complex expression, and it does not add or subtract zero. Furthermore, it tries to avoid multiplying by one, e. g., when computing the indices to an array access, if the array is defined as a byte array.

IV. DOP SCRIPTS

This section presents a selection of high-level exploit payloads implemented in SLANG. They not only demonstrate the capabilities of DOP and STEROIDS regarding expressiveness and complexity of payloads, but also security implications, because they feature ROP chain relocation and on-the-fly gadget search against targets *without* a built-in scripting engine.

Listing 5: Slang-code to compute the Levenshtein distance.

```
byte[]  s = "kitten"
byte[]  t = "sitting"

byte[56] d

func idx(int the_row, int the_col, int width)
    idx = the_row * width + the_col

func min(int a, int b, int c)
    min = a
    if(b < min)
        min = b
    if(c < min)
        min = c

for j from 0 to s.length + 1
    call idx(0, j, s.length + 1)
    d[idx] = j

for i from 0 to t.length
    call idx(i + 1, 0, s.length + 1)
    d[idx] = i + 1

    for j from 0 to s.length
        int j_plus_1 = j + 1

        call idx(i, j + 1, s.length + 1)
        int del_cost = d[idx] + 1

        call idx(i + 1, j, s.length + 1)
        int ins_cost = d[idx] + 1

        call idx(i, j, s.length + 1)
        int sub_cost = d[idx]
        if(s[j] != t[i])
            sub_cost += 1

        call min(del_cost, ins_cost, sub_const)
        call idx(i + 1, j + 1, s.length + 1)
        d[idx] = min
```

A. A classic algorithm: Levenshtein distance

We chose the Levenshtein distance [49] as an introductory example, because it is a well-known algorithm and allows to assess the compilation and execution times for the applications. It computes the edit distance between two strings, i. e.,

the number of operations necessary to transform one string into the other. In particular, our SLANG implementation in Listing 5 features: 1) Nested for-loops, requiring checking values for equality and conditional jumping. 2) 2D-array operations, i. e., memory access uses indices, which are computed using multiplication. 3) Numeric comparison and conditional execution to compute the minimum of three numbers. 4) Functions to compute minimum and indices.

B. SSL pointer-chain dereference

In OpenSSL, the data structure for saving an SSL private key is fairly complicated: From a base object (`ssl_ctx`), one has to successively follow a chain of eight pointers at different offsets in their respective data structures, to finally reach the secret key (see Listing 6). We adapted this example from Hu et. al [20], because it shows the use of a high-level description for a non-trivial BYOSE-attack. Naturally, not every application uses SSL private keys and such a base object may not always be withing immediate reach in every application. However, this is still a realistic attack, which is impossible without either an adaptive multi-step attack leaking data or a scripting engine. Note that one should not judge the script's complexity by it's brevity: Our experiments will show that executing it without optimization can require thousands of DOP gadgets.

Listing 6: Slang-code to retrieve a private key from SSL.

```
# The address of (or offset to) ssl_ctx is
# given as a parameter to the compiler.
addr p -> ssl_ctx

int[] offsets = [10 0 4 20 24 0 0]

for i from 0 to offsets.length
    p =* p
    p += offsets[i]

# At this point, p points to the private key and can
# be copied to an address of the attacker's choice.
```

C. ROP-chain Relocation

To account for ASLR, one can relocate a return-oriented programming chain (ROP chain), which means adjusting the addresses of the ROP gadgets (ROP gadgets) in the exploit buffer using a dynamically retrieved address. Attackers usually utilize a built-in scripting engine to perform this step, but this example accomplishes this task using DOP, demonstrating that this kind of attack can be applied to target applications without a built-in scripting engine. As Listing 7 shows, the attacker merely provides the address of a code pointer (`addr_of_code_pointer`), like a return address on the stack or a function pointer, and the offset between the code pointer and the image base (`offset_to_base`). Furthermore, she prepares the exploit buffer using the offset between the image base and the ROP gadgets, as it would be in a non-randomized address layout. Again, the brevity indicates rather SLANG's expressiveness than the script's complexity: It may very well require executing thousands of DOP gadgets.

Listing 7: Slang code to relocate a ROP chain.

```
byte[] rop_chain = {...}

# A ROP-chain holds data, mixed with addresses
# of ROP-gadgets. This array holds the offsets
# of those addresses in the ROP-chain buffer.
int gadget_offsets[] = [0x0C 0x14 0x24 0x28 0x2C]

addr image_base =* addr_of_code_pointer
image_base -= offset_to_base

# Relocate each gadget.
for i from 0 to gadget_offsets.length
    rop_chain[gadget_offsets[i]] += image_base
```

Listing 8: An exemplary ROP chain to execute `execve("/bin/sh")` in 32-bit Linux.

```
0x00: //bi                0x18: &buffer
0x04: n/sh                0x1C: &(pop ecx; ret)
0x08: 00 00 00 00         0x20: &buffer + 8
0x0C: &(pop eax; ret)     0x24: &(pop edx; ret)
0x10: 0B 00 00 00         0x28: &buffer + 8
0x14: &(pop ebx; ret)     0x2C: &(int 0x80)
```

D. Just-in-time return-oriented programming (JIT-ROP)

The relocation approach from the previous example does not work anymore, if a binary is protected with fine-grained code randomization such as *Binary Stirring* [24], because most of the previously known ROP gadgets are eliminated. Instead, an attacker can employ JIT-ROP to dynamically scan the process memory to search required gadgets on-the-fly, and assemble the ROP chain in the exploit buffer accordingly. Our implementation probes the program's code memory for specific gadgets and then uses their locations in a freshly generated ROP chain. Just like the other examples, the Slang script in Listing 9 can be compiled and executed for interactive and non-interactive mode DOP instances without changes.

Listing 9: Slang-code to scan the memory for ROP gadgets.

```
# pop eax, pop ebx, pop ecx, pop edx, int 0x80
byte[] g_bytes   = {58 c3 5b c3 59 c3 5a c3 cd 80}
int[]  g_lengths = [2 2 2 2 2]
int[]  g_offsets = [0x0C 0x14 0x24 0x28 0x2C]

addr start_addr =* addr_of_code_pointer

func is_gadget(addr a, byte[] gadget, int g_length)
    for i from 0 to g_length
        if(gadget[i] != a[i])
            return 0
    return 1

byte[] cur_g_bytes = g_bytes
for j from 0 to g_offsets.length
    addr at = start_addr
    loop
        call is_gadget(at, cur_g_bytes, g_lengths[j])
        if(is_gadget == 1)
            rop_chain[g_offsets[j]] = at
            break
        at += 1
    cur_g_bytes += g_lengths[j]
```

TABLE I: Comparison of the evaluated DOP-instances.
1: Implicitly, because the conditional command is present.
2: There is only one packet in memory at a time, thus one packet cannot modify the contents of the next packet.
3: Implicitly, through self-modifying DOP.
4: Synthesized from `mov` and `add` DOP gadget sequence.
5: Implicitly, through interactive mode/branch-free code.

	Interpreter	*Wireshark*	*GStreamer*	*Mini-Server*	*ProFTPd*
Arithmetic					
inc (*p += 1)	✗	✓[1]	✗	✗	✗
conditional inc					
(if(...) *p += 1)	✗	✓	✗	✗	✗
add (*p += *q)	✓	✗	✓	✓	✓
sub (*p -= *q)	✓	✗	✗	✗	✗
mul (*p $\star$= *q)	✓	✗	✗	✗	✗
gte (*p = *p $\geq$ *q)	✓	✗	✗	✗	✗
Movement					
mov (*p = *q)	✓	✓	✓	✓	✓[1]
conditional mov					
(if(...) *p = *q)	✗	✗	✗	✗	✓
load (*p = **q)	✗	✓	✓	✓[1]	✓
conditional load					
(if(...) *p = **q)	✗	✗	✗	✓	✗
store (**p = *q)	✗	✓	(✓)[3]	✓[2,4]	✓
Control-Flow					
goto	✓[1]	✓	✓	(✓)[5]	(✓)[5]
conditional goto	✓	✓	(✓)[3]	(✓)[5]	(✓)[5]
calculated goto	✓	✓	✗	(✓)[5]	(✓)[5]
Mode					
interactive	✗	✗	✗	✓	✓
self-modifying	✓	✗[2]	✓	✓[2,4]	✗

V. Applications with DOP Instances

We compiled and executed the example DOP scripts above for the five different applications we present in this section. Note that we only crafted the first of them: The other four were not "homemade" by us. Table I provides a comparison of the DOP instances with respect to the available DOP gadgets.

A. Interpreter

This bytecode interpreter is an exemplary application, which loads a file filled with bytecode into memory and then interprets the embedded instructions. Thus, this DOP instance is compatible with the non-interactive gadget chaining mode, although it does not require the exploitation of a memory error.

The interpreter has three arithmetic instructions: addition, subtraction, and multiplication. It also has an instruction to move data, and a `conditional goto` to modify the control-flow. Since the application internally uses an instruction-counter, both a `goto` and a `calculated goto` can be implemented by modifying its value.

The interpreter's comparably rich set of DOP gadgets is by far not complete: E. g., there are no comparison operators for (in)equality. Also, since the operand size is fixed to 32 bit, it is not immediately suitable to work on single bytes. Most importantly, this DOP instance lacks `load/store` gadgets.

Challenges. To interact with memory outside the DOP instance, we bootstrapped the `load/store` gadgets using *self-modifying DOP*. Since the data structures for all DOP gadgets are already in memory, an operation can alter the operands of other operations. We leverage a `mov` to overwrite the source or destination of another `mov` with values computed at runtime. Thus, we can simulate native `load/store` gadgets. Note that `mov` is the "weakest" of the three basic data movement DOP operations, and thus, any of the three can be used to synthesize the other two. This however, may not hold without self-modifying DOP. As we will see in the Section VI, this application shows that DOP gadgets such as `add` and `mul` are very important for the efficiency of the overall DOP-instance.

B. Wireshark

The packet analyzer Wireshark suffered from a stack-buffer overflow (CVE-2014-2299), which ultimately results in packet contents overflowing local variables (specifically `cinfo`) and parameters (specifically `packet_list`). Listing 10 shows the relevant lines for this application's DOP gadgets.

Detailed descriptions are available in the literature [20], [50], so we only give a brief summary of triggering the DOP gadgets here. Setting specific values to struct members of `packet_list` and `cinfo` enables `mov/load/store` and `conditional inc` DOP gadgets. However, the condition and the target of the `conditional inc` are not independent of one another, as they stem from the same value. Thus, one has to set up a fake `packet_list` data structure in memory, and use a `mov/inc/mov`-sequence to increment an arbitrary memory address. Additionally, a fake linked-list pointing at itself has to be employed to create an endless loop and keep the DOP instance running. Furthermore, the file position indicator serves as a virtual instruction pointer: manipulating its value results in a non-linear sequence of packets being read and enables a `goto`. The exploit payload is given at one go, but the data structures are being loaded to memory one by one. Thus, this DOP instance has properties of interactive and non-interactive modes: single-shot exploit, native `goto`, but no self-modifying DOP.

Listing 10: Partial Wireshark source code showing the lines for mov/load/store and conditional inc DOP gadgets.

```
record = packet_list->physical_rows[row];
record->col_text[col] =
        (gchar *) cinfo->col_data[col];
if(!record->col_text_len[col])
    ++packet_list->const_strings;
```

Challenges. Arithmetic operations are severely hampered since the only arithmetic DOP gadget is an `inc`, but expressive programs can still be compiled and the resulting DOP programs are efficient enough to pose a security threat.

To synthesize an `add` using the recipe given in Listing 2, we implemented an 8-bit `dec` using a 256-byte lookup table. This table is placed at a 256-byte aligned address, and its ith value is initialized to $i - 1 \bmod 256$. Now, we can write the value we want to decrement into the lowest byte of the address to the lookup table, and load the value saved there. The

implication of this construct is severe: We crafted an arithmetic DOP operation using only data-movement DOP operations.

C. Gstreamer

The Gstreamer multimedia framework suffered from a heap-corruption vulnerability (CESA-2016-0004) in its decoder for FLIC files, which are GIF-like animations. In particular, the vulnerability repeatedly allows to write arbitrary byte-sequences to arbitrary, positive offsets. The decoding happens in a fresh thread of a fresh process and since a thread's heap is usually aligned to 64 MB, the last three bytes of the addresses of multiple relevant heap objects are basically static. Evans published an exploit for this vulnerability and the write-up contains lots of technical details [51]. The author makes heavy use of partial pointer overwrites (namely, the last three bytes to keep the unknown first bytes intact) to modify the addresses used by two `memcpy`s to move data arbitrarily, to dereference pointers and even to abuse the frame-time calculation to perform addition. To do so, a fake `GstPad`-instance is created to avoid leaving the main loop, and data is copied into the input buffer to create new DOP gadgets, i. e., self-modifying DOP is used. Ultimately, a code-reuse attack is launched by crafting a call to the `system`-function. Listing 11 shows the details relevant for DOP.

Unlike the other applications, this one does not use a 32-bit address space, and thus shows that STEROIDS can also deal with 64-bit applications. Furthermore, this DOP instance does not rely on a stack corruption, but leverages a heap corruption, demonstrating that STEROIDS works across bug classes.

Challenges. While the exploit by Evans is certainly nifty, we have to go significantly beyond the published write-up.

In GStreamer, data is first read in 4 KB-sized chunks into a linked-list of `GstMemory`-objects. A class called `GstAdapter` is then responsible to merge these chunks of memory to accommodate the size the decoder actually needs. In this specific decoder, every frame is freshly allocated and therefore, their addresses become more unpredictable over time. Evans did not have to deal with this, because the entire exploit fit into the first 4 KB-chunk. However, reallocation is not acceptable for larger DOP programs: Our examples easily grew to 90 KB. Thus, we bootstrap our DOP programs by first copying their main content into the frame-buffer and then creating a fake `GstAdapter`-object to trick the application into thinking that there is only one big memory object. This not only facilitates reliable frame-wise self-modifying DOP, but also allows to modify the `skip-`, `size-` and `assembled_len`-variables of the `GstAdapter`-object to redirect DOP execution to arbitrary frames. Conceptually, this still belongs to the steps of gadget search and DOP instance setup to facilitate the execution of DOP gadgets, which is not in this paper's scope. We thus do not automate this task. In practice however, we used our flexible compilation and recipe mechanism to make it considerably easier. The second challenge is the addressing. The instructions that place constant data use relative offsets, but the `memcpy`s naturally require absolute addresses. Furthermore, the predictable addresses are all in the three byte-vicinity around the frame-buffer. Thus, a second bootstrap-step dynamically discloses the first bytes of the heap-objects and fixes the addresses of all indirectly addressed DOP-variables. Note that our compiler generates both these steps automatically.

As for the actual DOP gadgets, the `mov-`, `load-` and `add-`instructions were already described by Evans [51] and are rather straightforward. However, the arbitrary `store`-gadget was more challenging, because the native code does not contain the required data flow. We constructed the `store`-gadget by overwriting operands of a `mov`-gadget at run time utilizing self-modifying DOP. While `conditional goto` is not a strict requirement for Turing-complete computation, it helps to avoid unreasonable large payload sizes for complex DOP programs. We use the fact that the FLIC-decoder skips frames with the wrong header ID and we leverage the fake `GstAdapter` and self-modifying DOP to skip frames by modifying their header IDs in order to construct a `conditional goto`-gadget.

Listing 11: Partial GStreamer source code showing reading from an arbitrary address to the frame buffer's start (`dest`), placing of constant data at a chosen location in or after the frame buffer, writing to an arbitrary address, and the addition.

```
flx_decode_delta_fli(...) {
  ...
  // read; attacker controls content of flxdec
  memcpy(dest, flxdec->delta_data, flxdec->size);

  // attacker also controls contents of data...
  start_line = (data[0] + (data[1] << 8));
  lines =  (data[2] + (data[3] << 8));
  data += 4;
  dest += (flxdec->hdr.width * start_line);

 while(lines--) {
   ...
   count = *data++;
   while(count --) {
     ...
     // ... which determines offset and written value
     *dest++ = *data++;
  }
 }
}
...
gst_flxdec_chain(...) {
  ...
  while(...) {
    // calls flx_decode_delta_fli
    flx_decode_chunks(...);
    ...
    // write; attacker controls content of flxdec
    memcpy(flxdec->delta_data,
           flxdec->frame_data, flxdec->size);
    ...
    // add
    flxdec->next_time += flxdec_frame_time;
  }
  ...
}
```

D. Mini-Server

Hu et al. [20] modeled this application after an FTP-Server to serve as an example for DOP. However, please note that

it is in no way tailored to our compiler. The application suffers from a stack-based buffer overflow when reading data from a socket into a local buffer. Invoking the three DOP gadgets given in Listing 12 is fairly straightforward. However, one has to take care to preserve local variables, i. e., to reset `connect_limit` with each DOP-packet. Also, the parameter `buf` for `readData` is pushed onto the stack only once and needs to be restored each iteration.

Challenges. The `mov` DOP gadget cannot move all values, e.g., the value `STREAM` triggers another gadget. Thus, we opted to use the `load` to implement a general purpose `mov` for this application. We also had to use a temporary scratch space for the `conditional load`, because its `else`-case clobbers some adjacent values.

This DOP instance lacks the very important `store` gadget, which is necessary to write to arrays and other memory locations. Due to the interactive mode we cannot leverage self-modifying DOP to synthesize `store` from `mov/load`. Luckily, the `mov` and `add` DOP gadget are executed in sequence in a single packet. Thus, we can use the `mov` to modify the parameters of the `add` DOP gadget. This yields a store-add combination: `*(*p+4) += *q`, where the 4 stems from the different offsets for `typ` and `total`. Adjusting that offset leads to a `**p += *q` DOP gadget. To convert this into a pure `store`, we first use this DOP gadget to add the value at `**p` to itself, effectively multiplying it by two. In base two, this introduces a zero at the lowest bit. By repeating this 32 times, we can store a zero at a location of our choice. Finally, we invoke this DOP gadget a 33rd time, adding *q to the zero, which results in the desired `store`-DOP gadget (`**p = *q`). Again, the implication is severe: We used an arithmetic gadget (`add`) to achieve data-movement[2].

Listing 12: Partial Mini-Server source code showing the `conditional load`, `mov`, and `add` DOP gadgets. `size`, `srv`, `type` and `connect_limit` are local variables.

```
while(connect_limit--) {
  readData(sockfd, buf);     // stack buffer overflow
  if(*type == NONE) break;
  if(*type == STREAM)        // conditional load
      *size = *(srv->cur_max);
  else {
      srv->typ = *type;      // mov
      srv->total += *size;   // add
}}
```

E. ProFTPd

An integer overflow, which results in a stack-based buffer overflow, in the FTP server ProFTPd (CVE-2006-5815) allows the attacker to write almost arbitrarily into the process' memory. As a result, multiple DOP gadgets are available, which allow Turing-complete computation. There are six functions involved and the example is well-documented in the literature [20], [50], so we omit the vulnerable code. ProFTPd allows an interactive DOP-mode, but needs two packets per DOP gadget.

Challenges. Triggering the `conditional mov` gadget was straightforward. The `add` DOP gadget operates on a fixed global data structure, so one has to first move the two operands into said data structure, invoke the `add`, and then fetch the result. The `load` and `store` gadgets need to perform the double-dereferencing in two steps, which can be performed as a regular two-packet DOP gadget, but one needs additional `mov`s to place the parameters accordingly.

F. nginx

A flaw in the chunked size parser enables the attacker to inject a very large size number. Due to an unsafe unsigned to signed conversion that number can be interpreted as negative bypassing a buffer size check, which leads to a stack-based buffer overflow (CVE-2013-2028). Previous work [20] analyzed this bug, but ruled it out for DOP. We achieved a proof-of-concept DOP instance, albeit with limited capabilities.

Challenges. The target data structure `ngx_http_request_t` is used in many places, is therefore heavily constrained and clobbers many struct fields. We resolved this by rarely moving the struct in memory and fulfilling the constraints with memory preparation. Even though a `conditional load` and `byte inc` are available, the interactive mode impedes synthesizing `mov/store`. While this DOP instance can compute arithmetic operations and move values among the limited number of write slots, the `store`-constraint prevented us from running complex DOP scripts.

VI. Evaluation

To evaluate the capabilities of STEROIDS, we have compiled each of our four showcase DOP scripts against all five different applications from the last section. We then measured both compilation time and runtime, the size of the generated DOP program, and the number of executed DOP gadgets. Influences on runtime, results for optimized DOP scripts, and the number of necessary DOP gadgets are also discussed.

A. Quantitative Analysis

We conducted our experiments on Ubuntu 18.04.1 LTS, using a single core of an Intel i7 @ 2.9 GHz with 4 GB RAM.

For the Levenshtein algorithm, we measured the runtime of comparing a 7-character string against an 8-character string. Since not every application comes with SSL, we artificially added the relevant data structure into the program, and measured how long it took to dereference it. For the Relocator, we measured how long it took to relocate a given ROP chain. In our JIT-ROP example, however, the runtime of the algorithm depends heavily on where in memory the ROP gadgets are located. For comparable results, we thus normalized the algorithms to always scan exactly one kilobyte of memory.

Our results are summarized in Table II and Table III. The compile time scales roughly linear with the number of DOP gadgets. One DOP gadget takes about 150ms, which is

[2]Other constructs are possible, too. E. g., using only `dec` and `mul`: to move q to p, one multiplies p by 0, decrements it, multiplies it by -1 and finally multiplies it by q.

TABLE II: Evaluation of DOP programs compiled for three non-interactive applications: Interpreter, GStreamer, and Wireshark.
1: Optimizations: Loop unrolling; using variables instead of array.

		Levenshtein	SSL-Deref	Relocator	JIT-ROP	JIT-ROP[1]
	DOP gadgets	183	17	23	99	56
Interpreter	Compile Time	23.14s	1.53s	1.61s	12.11s	1.85s
	... per DOP gadget	0.12s	0.09s	0.07s	0.12s	0.03s
	Execution Time	0.29s	0.19s	0.20s	0.51s	0.49s
	... per DOP gadget	0.50μs	1898.89μs	2115.79μs	0.14μs	0.35μs
	Executed DOP gadgets	5855	99	95	35982	14180
	DOP gadgets	4407	1930	1419	2503	2689
Wireshark	Compile Time	179.75s	108.96s	91.62s	133.37s	107.89s
	... per DOP gadget	0.04s	0.06s	0.06s	0.05s	0.04s
	Execution Time	8.43s	0.75s	0.70s	77.85s	13.39s
	... per DOP gadget	2.81μs	3.59μs	3.84μs	2.91μs	2.85μs
	Executed DOP gadgets	2,994,468	210,989	195,758	26,733,658	4,691,568
	DOP gadgets	2284	265	317	740	1079
GStreamer	Compile Time	467.89s	59.72s	102.47s	227.07s	334.75s
	... per DOP gadget	0.20s	0.22s	0.36s	0.31s	0.31s
	Execution Time	16.33s	1.45s	1.59s	228.50s	149.58s
	... per DOP gadget	84.75μs	728.91μs	447.12μs	86.52μs	81.03μs
	Executed DOP gadgets	192,676	1,992	3,565	2,640,980	1,845,845

TABLE III: Evaluation of DOP programs compiled for two interactive applications: Mini-Server and ProFTPd.
1: Optimization: Loop-Check against constant.
2: Optimization: Loop unrolling; no branch-free transformation, since it is only one Basic Block.
3: Optimization: Using a single 16-bit-compare instead of two 8-bit compares. *: Estimated.

		Levenshtein	Levenstein[1]	SSL-Deref	SSL-Deref[2]	Relocator	Relocator[2]	JIT-ROP[3]	JIT-ROP[1]
	DOP gadgets	5466	4789	762	141	1112	37	594	316
Mini-Server	Compile Time	807.37s	686.47s	93.91s	17.68s	117.43s	10.86s	80.54s	50.40s
	... per DOP gadget	0.14s	0.14s	0.12s	0.12s	0.11s	0.29s	0.14s	0.16s
	Execution Time	5360.64s	420.36s	14.71s	0.62s	61.48s	0.25s	*22,956.00s	190.67s
	... per DOP gadget	1.28ms	1.28ms	1.39ms	4.46ms	1.28ms	6.83ms	*1.28ms	1.28ms
	Executed DOP gadgets	4,178,599	327,671	11,269	141	47,921	37	17,894,894	148,625
	DOP gadgets	2946	2813	354	126	607	72	501	195
ProFTPd	Compile Time	471.37s	447.27s	59.83s	23.56s	98.94s	21.74s	84.17s	33.15s
	... per DOP gadget	0.16s	0.16s	0.17s	0.19s	0.16s	0.30s	0.17s	0.17s
	Execution Time	3245.78s	1641.69s	6.93s	0.59s	49.21s	0.56s	*20,008.00s	123.95s
	... per DOP gadget	1.31ms	1.31ms	1.32ms	4.69ms	1.31ms	7.18ms	*1.31ms	1.31ms
	Executed DOP gadgets	2,472,032	1,250,336	5,243	126	37,452	72	15,238,408	96,406

mostly spent by the SMT solver to create the data structures. Naturally, more complex data structures have a tendency to take longer, especially when they are nested or when there are arrays involved. However, unless the available space for a data structure is very close to the minimal space necessary, we found the compile time to be fairly stable.

Note that this step can oftentimes be parallelized, if one decouples the compilation of single DOP gadgets. This can also be used in the interactive case, e. g., for GStreamer. However, since we evaluated on a single core, the additional overhead of starting separate processes may outweigh the benefit of simpler constraint solving. Furthermore, we noted that a noteworthy portion of a program's DOP gadgets are usually identical, which makes caching an attractive time saver during development.

A comparison of the number of executed DOP gadgets in Table II (e. g., for the Levenshtein algorithm) shows that the non-interactive mode enables efficient execution of DOP programs, even if the underlying application offers only minimal arithmetic capabilities. For the Interpreter, the runtime is even clearly dominated by the time to start the process (e. g., for the Relocator). However, a single DOP gadget in the interactive mode is roughly three orders of magnitude slower, because network-interaction dominates the runtime, despite using a local loopback interface in our experiments.

The DOP instance in Wireshark is non-interactive, but it reads the data structure for every single DOP gadget from a file. Hence, it spends a large amount of its runtime with file I/O. Similarly, GStreamer invokes costly `malloc`- and `memcpy`-operations for every DOP gadget, even before the actual gadget is executed. Nevertheless, the biggest impact of runtime stems from the lack of powerful DOP gadgets for arithmetic, in particular for equality comparisons and multiplications, which are utilized for array accesses. E. g. a subtraction operation would remedy this situation. This effect is most pronounced in the Wireshark example, because it does not even have an efficient `add`, which then has to be expressed through repeated `inc`.

While the execution time naturally has a certain variance, the chosen DOP programs are deterministic and there are many executed DOP gadgets. This seems to average out the execution time per DOP gadget in individual target applications,

especially since the runtime is usually dominated by factors other than computation.

These experiments show that STEROIDS facilitates the automatic compilation of DOP programs from a high-level language. Especially since the intermediary High-Level DOP-Asm and Low-Level DOP-Asm scripts are kept around, we argue that the generated programs can serve as an even more useful starting point for hand-crafted optimization.

If one reduces the number of the aforementioned problematic operations in SLANG, e. g, by using different variables instead of arrays or using pointer-arithmetic instead of array-indices to reduce multiplications, one can speed up the DOP program considerably (see last column of Table III). Especially when the number of loop iterations is known beforehand, the costly equality-check against a variable can be avoided. Instead one compares with a constant, which can often be implemented with a much cheaper conditional DOP operation (see last column of Table II). For a small and fixed number of iterations one can also unroll the loop to avoid the comparison altogether.

In the interactive gadget chaining mode, the attacker may not know the current state, e.g., specific values of variables, of the DOP instance during runtime. Consequently, computed branches are not supported, because the *Driver* on the attacker machine has no means to evaluate branch decisions and select the next DOP gadget. This branch-free transformation of the code has a huge performance impact. First, it increases the number of DOP operations. Second, loops cannot be exited early, because the attacker does not know whether the loop-condition is met and must therefore iterate, until an upper approximation of loop iterations is reached. This is especially critical for nested loops (e. g., in Levenshtein or JIT-ROP).

B. Implemented DOP operations

DOP operations fall into three separate categories: First, the DOP gadgets, which are implemented as gadget definitions, require the data-view of DOP. They are a little tricky to implement and they are highly specific to both the application and the vulnerability, so that it is unlikely that they can be reused. Luckily, one has to implement only very few of them. As Table IV shows, a new target application requires about five of these with an average length of 8.6 lines of code. GStreamer is the clear exception, because only the DOP operation to write constant data is implemented in this way.

Second, application-specific DOP operations, which are implemented in Low-Level DOP-Asm. They also have to deal with the oddities of the application, e. g, clobbered adjacent values, input values for which they do not work or application-specific variables. It is important, however, that these DOP operations do not export such behavior, i. e., they define clean High-Level DOP-Asm. Since the execution-view is more familiar, and since one deals with expected, albeit cumbersome behavior, they are much easier to implement. However, it is still unlikely that such a recipe can be reused for another application. There are again only about five of these DOP operations necessary to support a new application, because they usually only serve to correct the oddities of the underlying DOP gadgets. Wireshark in particular has to deal with this, which explains the high count of application-specific DOP operations. The exceptionally high line count for these DOP operations for the Mini-Server is caused by a single outlier: The `store-add-to-store`-gadget requires 72 lines.

TABLE IV: Number of implemented DOP operations. Different operand sizes were ignored for this table, as recipes for conversion are all reusable, and usually just use temporary variables to zero-extend or save parts of the operands.

	Interpreter	Wireshark	GStreamer	Mini-Server	ProFTPd
As Gadget definitions	6	5	1	5	5
∅ lines of code	3.8	14	9	5	11.2
App-specific	3	8	6	6	5
∅ lines of code	2.6	6.6	4.2	18.6	5.6
Reusable	5	11	11	16	11
∅ lines of code	6.2	10.5	8.3	4.7	4.6
Shared	5	5	8	13	11

Lastly, there are reusable DOP operations. They use only properly working DOP operations and are thus fairly easy to implement, despite using more lines. As Table IV shows, one needs roughly a dozen of these to execute all the DOP programs from our experiments on a particular target application.

More importantly, Table IV also shows that, if one keeps the distinction between these three categories in mind, e. g., with a naming convention to rule out accidental mix-ups, one can reuse many DOP operations. For example, four of the five target applications share the recipe for `mul` and `sub`. All in all, about 75 % of the DOP operations without application-specific oddities actually are reused. Also note the high count for the Mini-Server and ProFTPd. Both require the branch-free mode, which in turn requires gadgets that only have an effect, if the *here*-bit of a basic block is set. Not a single DOP operation from this category had to be implemented additionally to support ProFTPd.

In summary, to support four algorithms, we needed about 15-20 recipes per application, each having roughly 10 lines. While they may require domain knowledge to circumvent the app's peculiarities, writing them is (roughly) as simple as writing assembly. We found that about half of the recipes can be reused from other applications, and that they furthermore share certain patterns, which makes writing new recipes easier. Over time, we expect that more recipes can be reused and that therefore less recipes need to be implemented from scratch. Finally, note that an attacker is likely to only implement a very limited number of DOP gadgets for one specific payload.

VII. DISCUSSION

In this section, we discuss the security implications of the possibility to write complex DOP programs, limitations of our approach, and possible future work.

A. DOP Expressiveness

Our experiments have clearly shown that more powerful DOP instances lead to more efficient DOP programs. Still, it is also clear that remarkably little is necessary for DOP. E. g., our branch-free transformation not only shows that the interactive mode is just as expressive as the non-interactive mode, but also that neither a virtual program counter nor conditional DOP gadgets are necessary.

Furthermore, we have shown in the Interpreter example, that the data-movement DOP gadgets can express one another, i. e., that having either `mov` or `load` or `store` can be sufficient. The Wireshark example again lowers the bar for Turing-complete DOP by showing that one can express arithmetic gadgets through data-movement, while the MiniServer example shows that data-movement gadgets can express arithmetics. Lastly, none of our exemplary real-world target applications has a native logical gadget.

B. System Interaction

I/O is certainly useful, but we argue that arbitrary computations in another program's memory are useful on its own, e. g., to change the program's state or to compute values for further attack steps, like the DOP scripts in our evaluation. Naturally, DOP can only interact with the system, if the necessary system calls happen to be reachable via data flows. In this case, a recipe for a system call DOP gadget could be crafted. Alternatively, one would have to alter the control flow, i. e., use a recipe to wrap redirection, system call and return. While this could certainly be facilitated by DOP, one would cross the border of pure DOP. Similar to traditional code-reuse, an attacker is much more likely to use DOP to bootstrap an additional attack-phase to accomplish the chosen task in an easier way.

C. Security Implications

Our exemplary DOP scripts demonstrate how DOP and BYOSE can aid in leaking sensitive memory, in relocating a ROP chain to bypass coarse-grained code-layout randomization, and in finding ROP gadgets on-the-fly to bypass fine-grained code-layout randomization. However, there are other attacks against more powerful defense primitives, which also require a scripting environment. Thus, applications lacking a scripting environment are immune to those attacks, unless, of course, DOP is possible. E. g., one can bypass coarse-grained CFI [52] by probing the memory for ROP gadgets still usable even with the CFI policy in place. Evans et al. [53] bypassed the 64-bit version of code-pointer integrity (CPI), which protects sensitive pointers, by efficiently locating the safe-region it uses to hide its metadata. Furthermore, variants of XOM are vulnerable to runtime attacks deducting not-yet-protected bytes [54].

We believe the recent progression of DOA and DOP indicate that the research community should consider protecting data-flow and data structures when designing new systems defenses.

D. Limitations

Our research prototype has several limitations, which we want to discuss here together with their implications.

Data-Pointers. Our attacks may require a few data pointers or offsets as input, which may have to be leaked or guessed first. It can be argued that there are situations, where this is not excluded by the presence of ASLR (e. g., static modules, info leaks, process forks using the same memory layout, incompatible defenses, etc.), but it means that our exploits technically may not circumvent traditional ASLR. However, regarding code-reuse attacks, defenses like fine-grained code randomization [24] would be a stronger drop-in replacement for ASLR, but since they do not need to modify the data layout they are likely not to require such hard-to-guess data pointers. Furthermore, analogously to attacks targeting code-pointers [55], the GStreamer example shows that there are applications, in which relative offsets instead of absolute addresses suffice for DOP-attacks.

End-to-End Exploitation. While we deem this work to be a substantial improvement in automating DOP, it does not result in the fully automated construction of end-to-end DOP exploits. Firstly, the definitions of an application's DOP gadgets are not generated automatically. While we, conceptually, use the output of previous work [20], the format and details of said definition require manual effort. Secondly, embedding the data structure in the application's input and interfacing the application is left to the attacker, although our tools automatically generate the invocations to the interface.

Optimizations. Our prototype STEROIDS uses little traditional compiler optimization, like constant folding or common subexpressions reuse. However, in exploitation, a slightly better or smaller program may not merely be faster, but increase the chance of success substantially.

E. Future Work

Naturally, many technical limitations mentioned above can be seen as future work. However, we also want to discuss what we feel to be conceptual gaps in our understanding of DOP.

Confined and arbitrary code-execution. Further examination of the gap between DOP and arbitrary code-execution may be worthwhile. E. g., DOP may enable Turing-complete computation, but its results can not always be saved in a suitable format to interact with the outside of the DOP bubble.

Multi-Stage Bootstrapping. Our prototype does not reason about incomplete or constrained DOP gadgets, but instead relies on the attacker to create definitions or recipes for sufficient DOP gadgets. Similarly, the possibilities of executing multiple DOP gadgets in one step, like the `mov-add` to implement the `store` for Wireshark or the double-`mov` to implement the `store/load` for ProFTPd, are yet to be fully explored. An adaptive interactive mode, which uses info leaks to adapt the sequence or content of following DOP gadgets would also pose an interesting opportunity. If the search for DOP gadgets would support such bootstrapping, we suspect that many more applications would allow BYOSE-attacks.

Defenses. Concrete DOP attacks can be prevented in many ways: Fixing underlying vulnerabilities, data layout randomization, data obfuscation, or constrained read-/write-targets [12], [50]. Since DOP is arguably even more complex than ROP, there are likely even more subtle ways to hamper attackers. However, it remains a challenge to create efficient, easy-to-apply defenses, which constricts the very concept of DOP and DOAs.

VIII. Related Work

This section provides an overview over orthogonal attacks threatening modern defense mechanisms, the state of the art for data-only attacks, and defenses trying to prevent data-only attacks.

A. Orthogonal Attacks

Counterfeit-object oriented programming (COOP) [56] achieves ROP-like capabilities using only full-function gadgets comprised of the methods of suitably crafted objects, without violating the non-C++-specific Control-flow graph (CFG). Crash Resistance [57] allows to probe memory, even though a careless probe should crash the application. In combination with a full-function reuse technique, this can thwart code randomization and schemes for information hiding. Blind ROP [30] uses a side-channel to locate gadgets in an otherwise unknown binary with a stack-buffer vulnerability. This requires that the attacked server-application is restarted after a crash, without being rerandomized. *Control-Flow Bending* [58] is a code-reuse attack that exploits the coarse-grained nature of some CFI implementations. They chain pairs of calls to standard library functions such as `memcpy/printf` and control their arguments to achieve Turing-complete computation on the data plane.

Conti et al. [59] show that current CFI- and Shadow Stack-implementations are vulnerable to attackers controlling stack-values though a heap-based vulnerability. Göktas et al. [52] show that coarse-grained CFI does not prevent the execution of call-site gadgets and entry-point gadgets. Control Jujutsu [60] can circumvent even fine-grained CFI with a shadow stack, because modern applications often use coding practices, which exceed the limits of current pointer analysis. Snow et al. [61] presented multiple attacks highlighting implementation pitfalls of destructive code-read (DCR)-based defenses, whereas BGDX [54] shows a more general attack deducing the location of not-yet-protected gadgets.

Q [48] is conceptually similar to this work in that they provide a language for exploit programming, too. The key difference is that a) their language is more low-level than SLANG, b) their languages is based on ROP gadgets, which are much more regular across applications and easier to reuse.

The authors of *Microgadgets* [62] define classes of ROP gadgets, and, trying to use those members with the fewest bytes, implement higher-level operations. E. g., using multiple `xor`-gadgets to implement a `store`, which is similar in concept to our recipes, but without supporting the automatic selection or recombination we achieve.

B. Data-Only Attacks

Chen et al. [11] have shown that attacking non-control-data can have consequences just as severe as code-reuse attacks. *Memory Cartography* [45] is an automatic way to create a net of memory references to navigate reliably through data structures, and can be used it to create data-only exploits (DOE), which are robust to ASLR. Hu et al. presented a technique called *dataflow stitching* [22], which analyses potentially corruptible data-flow in an application, especially with regard on how to chain individual data-flows. However, their attacks are targeted to enable specific attacks targeting the peculiarities of a particular application, much in the spirit of Chen et al.. Based on these stitching techniques, however, they recently developed what is now called DOP [20]. The authors not only show that DOP often allows Turing-complete computation, but also that DOP gadgets are frequently available. However, they focus mainly on gadget search, and leave both the DOP instance setup and the payload preparation to manual work.

Block-Oriented Programming [21] accepts a script in a high-level language and a read/write-primitive as input. In a nutshell, it then tries to solve the constraints a given program has, to ultimately synthesize the script using data-oriented techniques. It does so with a high degree of automation, but the authors also show that their approach of program synthesis via constraint solving is NP-hard. Thus, the synthesized programs are rather simple, essentially only making sure that some data ends up in a sensitive sink. We would argue that these programs still belong to the step of DOP instance setup.

In contrast, this paper is mostly not concerned with DOP *gadget search*, or the *DOP instance setup*, but instead focuses on the *payload preparation*, which is automating the process of creating the data structures that drive an application to execute the many instructions of a DOP program.

C. Defenses against Data-Oriented Attacks

Current countermeasures against data-flow attacks seem to mirror those against control-flow attacks. Lin et al. [19] and *SALADS* [18] randomize the layout of data structures. Both *data-randomization* [16] and *Data Space Randomization* [17] masks data in-between uses by XOR-ing them with random values. The authors of *HARD* [15] take this concept to the hardware-level, with a customized RISC-V architecture. These five techniques are based on introducing randomness, but may not protect all data and furthermore require source code, which makes them unsuitable for protecting commercial off-the-shelf (COTS) binaries. *ValueGuard* [14] inserts canary values in-between buffers and other data to detect a buffer overflow. Bhatkar et al. [63], shuffle functions, randomize heap object-locations and location of the stack, and use separate stacks for stack buffers. Both approaches significantly hamper DOAs, but may not protect against such attacks in all cases (especially if they are not based on buffer overflows).

Analogous to CFI, Data-flow integrity (DFI) [13] uses a statically determined data-flow graph to restrict the data-flow at runtime. Write-integrity testing (WIT) [12] analyses a program to color instructions and memory objects. It then

ensures at runtime that instructions only write to objects with a matching color. *CUP* [64] combines memory-maps with fat-pointers to ensure spatial and (probabilistic) temporal memory safety, even for stack-variables. *HardScope* [50] carries the concept of a variable's visibility-scope from the source code to the runtime. These techniques incur high overhead, even despite the fact that *HardScope* is implemented with hardware support on an open-source microcontroller.

Exploits are naturally brittle, so we would expect most defenses to thwart most unmodified DOP programs. In comparison to code-flow transfers, data is very dynamic and touched virtually everywhere in a program, which makes it hard to reason about and inefficient to check its integrity. Thus, we expect that it will be even harder to create efficient defenses, which prevent DOAs in principle rather than by chance. E. g., the authors of *HardScope* acknowledge that preventing the advanced exploit against GStreamer [51] that we also use in our experiments is more challenging, because it corrupts a heap object it could use legitimately as well. Similar to CFI bypasses, some DOAs may fall through the cracks, e. g., due to unprotected modules, false metadata, or coarse class-granularity.

IX. Conclusion

In the last decade code-reuse defenses gained a lot of attention in academia and industry, which led to their wide-spread adoption. However, the mitigations of the orthogonal DOA and DOP attacks are still in the early stages of development.

In this paper we demonstrate that even constrained DOP instances can be escalated in expressiveness to execute complex DOP programs. We show that DOP programs aid in bypassing code-reuse defenses to launch advanced code-reuse attacks such as JIT-ROP. This BYOSE feature of DOP transfers a whole class of just-in-time attacks to targets without a built-in scripting engine. Our high-level SLANG language can be used to implement portable DOP exploits and our DOP compiler STEROIDS automatically constructs the required low-level data structures to run them in target applications. We hope that our results raise the awareness for DOA and DOP in the research community and aid in the development of systems defenses that consider the mitigation of these attacks.

Acknowledgment

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (ERC Starting Grant No. 640110 BASTION). In addition, this work was supported by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0592K HWSec).

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, 2009.

[2] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in *USENIX Security Symposium*, 2013.

[3] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *IEEE Symposium on Security and Privacy*, 2013.

[4] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code," in *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[5] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities," in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[6] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[7] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-Execute-After-Read: Preventing Code Disclosure in Commodity Software," in *ACM Asia Conference on Computer and Communications Security (ASIA-CCS)*, 2016.

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

[9] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical Code Randomization Resilient to Memory Disclosure," in *IEEE Symposium on Security and Privacy*, 2015.

[10] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A. Sadeghi, T. Holz, B. D. Sutter, and M. Franz, "It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[11] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data Attacks Are Realistic Threats," in *USENIX Security Symposium*, 2005.

[12] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *IEEE Symposium on Security and Privacy*, 2008.

[13] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2006.

[14] S. V. Acker, N. Nikiforakis, P. Philippaerts, Y. Younan, and F. Piessens, "Valueguard: Protection of native applications against data-only buffer overflows," in *Information Systems Security - International Conference (ICISS)*, 2010.

[15] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek, and M. Franz, "Hardware assisted randomization of data," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2018.

[16] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, "Data randomization," Technical Report TR-2008-120, Microsoft Research, 2008. Cited on, Tech. Rep., 2008.

[17] S. Bhatkar and R. Sekar, "Data space randomization," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[18] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu, "A practical approach for adaptive data structure layout randomization," in *European Symposium on Research in Computer Security (ESORICS)*, 2015.

[19] Z. Lin, R. D. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2009.

[20] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *IEEE Symposium on Security and Privacy*, 2016.

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

[22] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *USENIX Security Symposium*, 2015.

[23] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated Software Diversity," in *IEEE Symposium on Security and Privacy*, 2014.

[24] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[25] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *IEEE Symposium on Security and Privacy*, 2018.

[26] C0ntex, "Bypassing non-executable-stack during exploitation using return-to-libc," http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf, 2010.

[27] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, 2012.

[28] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *ACM Asia Conference on Computer and Communications Security (ASIA-CCS)*, 2011.

[29] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, 2012.

[30] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *IEEE Symposium on Security and Privacy*, 2014.

[31] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[32] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," in *Annual Computer Security Applications Conference (ACSAC)*, 2006.

[33] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *IEEE Symposium on Security and Privacy*, 2012.

[34] P. Team, "PaX address space layout randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt.

[35] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely Rerandomization for Mitigating Memory Disclosures," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[36] L. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can - secure and efficient ad-hoc instruction-level randomization for x86 and ARM," in *ACM Asia Conference on Computer and Communications Security (ASIA-CCS)*, 2013.

[37] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *USENIX Security Symposium*, 2012.

[38] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, "Marlin: A fine grained randomization approach to defend against ROP attacks," in *International Conference on Network and System Security (NSS)*, 2013.

[39] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *IEEE Symposium on Security and Privacy*, 2013.

[40] ——, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE Symposium on Security and Privacy*, 2013.

[41] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming," in *Symposium on Network and Distributed System Security (NDSS)*, 2015.

[42] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and Deployable Continuous Code Re-Randomization," in *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[43] M. Backes and S. Nürnberger, "Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing," in *USENIX Security Symposium*, 2014.

[44] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[47] R. Tarjan, "Testing flow graph reducibility," in *ACM Symposium on Theory of Computing*, 1973.

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

[46] R. Rojas, "How to make zuse's z3 a universal computer," *IEEE Annals of the History of Computing*, 1998.

[48] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *USENIX Security Symposium*, 2011.

[49] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[50] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A. Sadeghi, "Hardscope: Thwarting DOP with hardware-assisted run-time scope enforcement," 2017. [Online]. Available: http://arxiv.org/abs/1705.10295

[51] C. Evans, "Advancing exploitation: a scriptless 0day exploit against linux desktops," https://scarybeastsecurity.blogspot.de/2016/11/0day-exploit-advancing-exploitation.html, Nov. 2016.

[52] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of Control: Overcoming Control-Flow Integrity," in *IEEE Symposium on Security and Privacy*, 2014.

[53] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the Point(er): On the Effectiveness of Code Pointer Integrity," in *IEEE Symposium on Security and Privacy*, 2015.

[54] J. Pewny, P. Koppe, L. Davi, and T. Holz, "Breaking and fixing destructive code read defenses," in *Annual Computer Security Applications Conference (ACSAC)*, 2017.

[55] E. Göktaş, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida, "Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, Apr. 2018.

[56] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. R. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *IEEE Symposium on Security and Privacy*, 2015.

[57] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding," in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

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

[59] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[60] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[61] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis, "Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks," in *IEEE Symposium on Security and Privacy*, 2016.

[62] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz, "Microgadgets: Size does matter in turing-complete return-oriented programming." in *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[63] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *USENIX Security Symposium*, 2005.

[64] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cup: Comprehensive user-space protection for c/c++," in *ACM Asia Conference on Computer and Communications Security (ASIA-CCS)*, 2018.