PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution

Andreas Follner[1], Alexandre Bartel[1], Hui Peng[2], Yu-Chen Chang[2], Kyriakos Ispoglou[2], Mathias Payer[2], and Eric Bodden[3]

[1] Technische Universität Darmstadt, Darmstadt, Germany
andreas.follner@cased.de, alexandre.bartel@cased.de
[2] Purdue University, West Lafayette, USA
peng124@purdue.edu, chang397@purdue.edu, kispoglo@purdue.edu, mathias.payer@nebelwelt.net,
[3] Paderborn University & Fraunhofer IEM, Paderborn, Germany
bodden@acm.org

Abstract. Return-Oriented Programming (ROP) is the cornerstone of today's exploits. Yet, building ROP chains is predominantly a manual task, enjoying limited tool support. Many of the available tools contain bugs, are not tailored to the needs of exploit development in the real world and do not offer practical support to analysts, which is why they are seldom used for any tasks beyond gadget discovery.
We present PSHAPE (*P*ractical *S*upport for *H*alf-*A*utomated *P*rogram *E*xploitation), a tool which assists analysts in exploit development. It discovers gadgets, chains gadgets together, and ensures that side effects such as register dereferences do not crash the program. Furthermore, we introduce the notion of *gadget summaries*, a compact representation of the effects a gadget or a chain of gadgets has on memory and registers. These semantic summaries enable analysts to quickly determine the usefulness of long, complex gadgets that use a lot of aliasing or involve memory accesses. Case studies on nine real binaries representing 147 MiB of code show PSHAPE's usefulness: it automatically builds usable ROP chains for nine out of eleven scenarios.

1 Introduction

Exploiting software vulnerabilities was simple and straightforward up until the early 2000s, when mitigation techniques were scarce and seldom applied. In contrast, contemporary systems deploy a multitude of defense mechanisms such as stack canaries [5], data execution prevention (DEP) [1], and address space layout randomization (ASLR) [18], each of which presents an obstacle to exploitation that needs to be bypassed. This has largely restricted exploit development to manual effort with only basic tool support.

While the circumvention of mitigations has been studied in detail, there is no comprehensive and automatic approach to bypassing all mitigations at once. Different mitigations must be defeated using different attacks. For example, DEP

is bypassed using ROP or other code-reuse attacks [4, 6, 17, 26, 32]. Although DEP exploits may share some commonality amongst themselves, this does not carry over to exploits targeting ASLR and stack canaries, which tend to be very scenario-specific and often rely on another vulnerability in addition to the one that allows a code pointer to be overwritten. In general, information leaks [2,7,31] are the preferred way of learning a program's memory layout and contents. However, these often require either another vulnerability like a format string vulnerability, or a scriptable environment under the analyst's control such as Javascript or Actionscript. More sophisticated attacks must operate with stricter constraints or are limited to a specific use case [3, 12, 19, 29, 33].

Current exploits consist of three stages: (i) information collection to bypass ASLR, (ii) ROP to bypass DEP, and (iii) executing the desired payload. The first stage uses an information leak to discover all required information to get around ASLR. The second stage uses ROP to initialize a memory area and remap it as executable. Then, in the third stage, the exploit runs classic shellcode within the newly mapped region. Exploits are split into three stages because (i) information leaks are program specific and (ii) ROP programming is cumbersome, complicated, and hard to control. Attackers prefer short ROP chains and inject and execute binary code as soon as possible.

The plurality of mitigations complicates the automation of exploit generation, yet certain mundane tasks, particularly those in later stages of ROP chain creation, are good targets for automation. These tasks include finding gadgets, assessing gadgets' usefulness, and combining gadgets to achieve useful behavior. These tasks require the analyst to (i) decompose the code she wants to execute into analogues of individual assembly instructions (e.g., write a certain value in a register), then (ii) manually find individual gadgets whose semantics correspond to all the individual assembly instructions, (iii) undo any unintended side effects of executed gadgets, and (iv) ensure that preconditions, such as that a register has to point to writeable memory, are satisfied. While many tools have been proposed to automate these steps, every single one appears to show at least some serious limitation when it comes to practical application scenarios.

This work presents and evaluates PSHAPE, a novel approach to automatically perform steps (i) through (iv) through a semantic gadget search and gadget summaries. We assume that the analyst wants to execute a function to make her payload executable, following the idea of three-stage exploit development. First, PSHAPE discovers all gadgets in a given binary and computes their pre- and postconditions. Afterwards, PSHAPE selects the best suited gadgets for loading or modifying values in registers used for passing arguments to functions. Second, PSHAPE combines these gadgets into chains to create a chain of non-interfering gadgets. Finally, in the third step, gadgets may be added to the chain to make sure that the analyst can initialize all registers the chain dereferences, as its execution may otherwise lead to a crash.

This work compares PSHAPE to twelve other tools. Since no other tool offers gadget summaries, one can only compare the number of gadgets found and how well the gadget chaining mechanisms work. For the latter, we use four

Linux and five Windows binaries, a total of 147 MiB of executable data. Then, we use the tools to create gadget chains that initialize between three and six registers with analyst-controlled data, allowing the analyst to invoke functions often used in ROP exploits, such as `mprotect`, `mmap`, or `VirtualProtect`. This results in eleven scenarios, for which a gadget chain can be created.

To summarize, the work presents the following original contributions:

- gadget summaries, a compact view on a gadget's semantics, greatly enhancing the search for useful gadgets,
- a mechanism to automatically generate a gadget chain that initializes registers used for passing parameters to execute an arbitrary method, making sure that all preconditions are satisfied,
- PSHAPE, an open-source implementation of the approach, and an evaluation of PSHAPE comparing it to other ROP tools. We show that it can automatically produce chains for nine out of eleven scenarios (81%), passing up to six parameters to function calls, while other tools can create a chain only in one scenario.

2 Motivating Example

In this section we show how PSHAPE helps building exploits for real-world vulnerabilities. The example we use is CVE-2013-2028, a typical buffer overflow vulnerability, which was found in the *nginx* web engine[4].

For our running example, the goal is to inject arbitrary shellcode, make it executable, and then overwrite the return address with the beginning of the shellcode. To bypass DEP, `mprotect` needs to be called using a ROP chain to make the shellcode page executable. This includes performing the following tasks: (1) information leaking, to discover the address of `mprotect` and the stack frame where the vulnerable buffer is allocated, (2) building the ROP payload for calling `mprotect`, and (3) constructing the shell code.

Problem Definition While all the tasks mentioned above are difficult, the second task can become increasingly complex due to the huge number of gadgets and constraints that need to be tracked along the gadget chain. Manually crafting the payload is both time-consuming and tedious. To execute a system call or call any other function, an attacker must (i) identify all the registers needed to be initialized, e.g., setting up the syscall number in `rax` and preparing arguments in other registers; (ii) for each register to be initialized, search all the relevant gadgets in the binary using a gadget finding tool; (iii) analyze each gadget to find out how it affects registers and memory; (iv) choose a subset of the identified gadgets and chain them into a coherent exploit.

The task of finding appropriate gadgets for initializing a register in steps (ii) and (iii) is complex and takes a very long time for the analyst. For example,

[4] `http://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html`

running ROPgadget [27] on nginx with a `grep` filter to identify gadgets for touching `rax` produces a list with thousands of candidates. Since ROP gadgets often have unwanted side effects on other memory locations and/or registers (e.g., writing to invalid addresses and causing a crash), only a few gadgets remain viable. This makes the identification of usable gadgets a slow process, even for an experienced analyst. Moreover, chaining gadgets in step (iv) is a repetitive task whose complexity entails a lot of work. To execute a system call using ROP, all arguments need to be passed beforehand, involving memory writes, register initialization and typically cannot be done with only a single gadget. Thus, finding gadgets for different operations is crucial for building the payload. Since finding different gadgets requires iterating through the process of gathering and filtering the viable gadgets, the more gadgets a ROP chain needs, the more heavy manual workload is required.

Our Approach The automation provided by PSHAPE can simplify the last three steps significantly, saving the analyst from large volumes of repetitive work. For steps (ii) and (iii), PSHAPE can assist in two ways. First, it reduces the result set size of a gadget search by filtering out incompatible gadgets, such as arithmetic gadgets. Second, it produces gadget summaries that speed up the process of gadget analysis. PSHAPE greatly reduces the amount of manual work required in step (iv) by chaining gadgets into an exploit completely automatically, whilst also satisfying any preconditions of the constituent gadgets.

3 Automating Exploit Generation

PSHAPE assists an analyst during exploit development by offering two distinct features which set it apart from existing tools that are publicly available, namely it (i) provides summaries based on gadget semantics, making it straightforward for an analyst to assess and select gadgets, and (ii) chains gadgets together so that they load registers used to pass parameters to functions with analyst-controlled data. This allows the invocation of arbitrary functions. PSHAPE also ensures that any preconditions of a gadget (such as that a register has to point to readable memory) are satisfied.

We first define what gadget summaries are and how they are computed in § 3.1 and then describe our approach to generate gadget chains in § 3.2.

3.1 Gadget Summaries

Overview ROP mitigations that (i) monitor program executions and detect short code sequences [8, 9, 14, 25] or (ii) require all `return` operations to return to an instruction following a call instruction [25, 39] force developers into using long gadgets or even entire functions [29]. The increasing length of gadgets makes manual analysis and reasoning increasingly difficult. We thus propose *gadget summaries*, which reflect a gadget's semantics in a compact specification that allows analysts to understand a gadget's behavior at a glance. Fig. 1 shows an

```
mov rax, rsp
mov [rax+20h], r9
mov [rax+18h], r8
mov [rax+10h], rdx
mov [rax+8], rcx
mov rcx, r9
mov rax, [rcx]
inc rax
mov [rcx+8], rax
mov rax, [rcx+4]
inc rax
mov [rcx+0Ch], rax
ret
```

(a) A candidate gadget.

```
PRE: [r9] <-> [r9 + 0xC]
PRE: [rsp] <-> [rsp + 0x20]
POST: rsp = rsp + 8
POST: rax = [r9 + 4] + 1
POST: rcx = r9
```

(b) Corresponding gadget summary.

Fig. 1: Despite this being a relatively short gadget in `mshtml.dll` which contains only 13 instructions (a), analyzing it manually is still a cumbersome and error-prone task. PSHAPE automates this process by creating a simple summary (b). Note that by default PSHAPE does not display memory write postconditions as they are seldom of interest, and make the summary harder to read.

example of a gadget summary, with the gadget on the left, and its summary on the right. This gadget has two preconditions, because `r9` and `rsp` are dereferenced. The actual effects on the program state are that `rsp` is increased by 8, `rax` receives the value of 1 + `[r9 + 4]`, and `rcx` is assigned the value of `r9`.

Method First, gadgets are identified by finding `return` opcodes and backward disassembly. These gadgets are then converted into an *intermediate representation* (IR) to simplify analysis. Our current prototype uses VEX IR, see § 3.3. Based on this IR, PSHAPE propagates all assignments, such as to temporary or real registers, or memory locations forwards, resulting in a single statement for each real register and memory location. This single statement (referred to as *postcondition*) contains all operations on this register or location, i.e., an abstraction of the new value after a gadget has executed. Of course our analysis models memory locations so it is able to correctly determine postconditions of gadgets that use the stack to pass data. E.g., it detects that after a `push rax ; pop rbx ; ret` gadget, `rbx` contains the value of `rax`. This analysis also allows us to readily extract *preconditions*, such as register or memory dereferences. Post- and preconditions combined result in a *gadget summary*, a compact representation of the state of memory and registers after a gadget has executed along with a list of dereferenced registers and offsets. Our syntax for pre- and postconditions is similar to assembly syntax, and should be intuitive for binary analysts. The current prototype excludes instructions such as jumps, loops, or bit manipulation in the summaries to reduce the explosion in state and complexity, see § 6. We leave more involved search strategies for future work.

As memory is often accessed sequentially using offsets from a register, one can compress summaries by merging such accesses into a range. For example, preconditions `[rax]`, `[rax + 8]`, `[rax + 0x10]` and `[rax + 0x20]` can be compressed

to: `[rax] <-> [rax + 0x20]`. This denotes that all memory between `[rax]` and `[rax + 0x20]` has to be read/writeable. This heuristic sacrifices precision, as not every single byte must be accessed, but makes summaries concise.

Gadget summaries aid the analyst in the process of understanding how a gadget affects the state of registers and memory and are increasingly helpful, the more instructions and aliasing a gadget contains. They also allow for a more efficient gadget search, as expressing postconditions when searching for a gadget is much more intuitive and flexible than specifying a certain instruction. Lastly, gadget summaries are useful for selecting gadgets for automated gadget chain generation, which we describe in the next section.

3.2 Gadget Chaining

Our approach aims at finding a valid and short gadget chain which loads analyst-controlled data, i.e., relative to `rsp`, into registers. This allows invoking an arbitrary function with analyst-specified parameters. It consists of three steps, as shown in Fig. 2. In the first step, the gadgets are extracted from the target binary and summaries are computed. Then, based on the summaries, the list of gadgets are filtered to keep only the ones related to initializing registers that are used for passing function parameters. The second step combines these gadgets into chains. For a chain, pre- and postconditions are computed, and if the chain has the desired postconditions, the third step analyzes the validity of each chain and adds gadgets to satisfy any preconditions.

Step 1: Gadget Extraction and Summary Computation. First, gadgets are extracted from a given binary, delivering a list of gadgets for which we then compute gadget summaries. The results are stored, making them available for the analyst. Next, the gadgets are filtered to keep only the ones related to initializing registers used for passing parameters to functions. On 64-bit Windows those are `rcx`, `rdx`, `r8`, and `r9`, in that order. On 64-bit Linux the registers used for parameter passing to functions are `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`, in that order. Additional parameters are passed on the stack in both cases. Our summaries simplify filtering because gadgets that do not set the registers stated above to a value that can be controlled by the analyst, are discarded automatically.

We divide these gadgets into two categories, *load* and *mod*. Gadgets in the load category overwrite a given register, e.g., a `pop` instruction, while gadgets in the mod category modify it, e.g., an `add` instruction. Gadgets in the load category are favored, and within this category, gadgets that use `rsp`-relative memory dereferences are preferred, as `rsp` needs to be under the control of the analyst anyway when using ROP. For example, a `pop rcx` gadget is preferred over a `mov rcx, [rax]` gadget. If no suitable *load* gadgets exist, *mod* gadgets such as `add rcx, rax` are used. Based on this ranking and the number and severity of pre-, and postconditions, the n most suitable gadgets for loading each parameter register with arbitrary data are selected and passed to Step 2. The step of assessing the severity of pre- and postconditions reuses some ideas presented in GaLity [13].

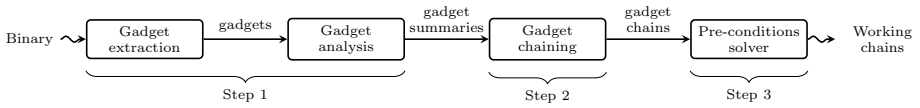

Fig. 2: Overview of our approach to generate gadget chains. In Step 1, we extract gadgets up to a certain size and create summaries. In Step 2 we select a set of n gadgets for the individual parameters to constrain the search space and then create gadget chains. In Step 3 we analyze the chains and prepend gadgets to make any dereferences analyst-controlled.

Step 2: Combining Gadgets into Chains. In the second step, the gadgets from Step 1 are combined and all possible permutations of a chain are computed. Remember that in Step 1, the n most suitable gadgets are selected for every parameter register. E.g., invoking a function with four parameters results in $n^4 \times 4!$ possible chains. For each permutation of a chain, pre- and postconditions of the whole chain are computed. If a chain's postconditions are not the expected result, i.e., the registers used to pass parameters do not contain analyst-controlled data, it is discarded. Instead of exhausting the search space, we stop the exploration after the first viable combination is found.

Step 3: Solving Pre-Conditions. It may happen that a chain generated in Step 2 contains preconditions such as register dereferences. The analyst needs to have the possibility to initialize the dereferenced registers, so they contain the address of a valid memory area. In Step 3, PSHAPE attempts to build a gadget chain that allows loading analyst-controlled data into an arbitrary register. Once such a gadget is found it is prepended to the incoming chain, forming a new chain. The new chain is then checked for pre- and postconditions again to make sure it does indeed initialize dereferenced registers and does not interfere with the original chain. Note that the number of iterations is limited (four in our prototype), so the chain does not grow forever.

Our gadget chaining fully automates the process of stitching gadgets together to initialize registers used for passing parameters to functions with data the analyst controls. It also adds gadgets to the chain to ensure any dereferenced registers are also initialized with data the analyst controls. This approach simplifies exploit development, especially if functions taking many parameters are called or if the available gadgets consist of many instructions.

3.3 Implementation

PSHAPE uses a standard technique to discover gadgets: first, using pyelftools[5] and pefile[6], it finds executable sections in an input binary. Afterwards it scans these sections forwards bytewise until a return opcode is found, storing these

[5] `https://github.com/eliben/pyelftools`
[6] `https://github.com/erocarrera/pefile`

offsets in a list. Then, using several threads, it disassembles backwards from these offsets using the Capstone framework [22]. To limit the number and complexity of gadgets and speed up the discovery process, the analyst can specify the minimum and maximum size, i.e., number of instructions, of a gadget.

If the disassembly yields only legal instructions, we convert this gadget to Valgrind's VEX IR [21] using PyVEX [34]. Lifting the original assembly code to VEX has the advantage that it is much simpler to analyze because there are fewer instructions and side effects are made explicit. After this conversion, VEX assignments are propagated forward, resulting in a single statement for each real register and memory location, which contains all operations on this register or location, i.e., an abstraction of the new value after a gadget has executed.

4 Evaluation

We first compare PSHAPE with existing tools regarding their ability to extract gadgets from binaries as well as their ability to construct gadget chains to initialize registers for function calls in Section § 4.1. Then, in Section § 4.2, we qualitatively evaluate gadget chains that PSHAPE creates for various binaries, and discuss optimizations.

4.1 Comparison with Existing Tools

In Table 1 we have listed the tools designed to help an analyst to create ROP exploits. Generally, we have found that there is a big gap between the theoretical state of the art and what actually exists and works well in practice. Many of the tools we evaluate contain bugs and other quirks that limit their usefulness in real scenarios, the main focus of PSHAPE.

OptiROP and Q are not publicly available and also were not made available to us upon request. We also excluded nrop due to its scope (see § 5). We managed to compile ROPC although it has been unmaintained for years, and GitHub issue reports are not answered. Unfortunately, it could not extract gadgets from any of the binaries we use in the evaluation, which is why we exclude it.

For the evaluation we use five Windows binaries: firefox.exe, iexplore.exe, chrome.exe, mshtml.dll, and jfxwebkit.dll, and four Linux binaries: chromium, apache2, openssl, and nginx, representing a total of 147 MiB of executable data. Detailed information about the binaries and PSHAPE are available on the companion website: `https://sites.google.com/site/exploitdevpshape/`.

Gadget Discovery In this section, we compare the different gadget discovery routines. For a tool to be considered in these experiments, we require that it can read ELF or PE binaries and find gadgets in 64-bit binaries. DEPlib, Agafi, mona.py, Ropeme, and MSFrop do not fulfill these requirements and were therefore discarded, leaving us with the following tools to compare to: ROPgadget, rp++ and ropper. We configured them to look for gadgets up to a maximum length of 35 instructions. Table 2a summarizes the results.

Tool		Syntactic Search	Semantic search	Gadget Chaining	Turing Complete	Open-Source	Binary available	PE	ELF	64-bit
PSHAPE		✓	✓	✓	×	✓	✓	✓	✓	✓
OptiROP [23]		✓	✓	✓	×	×	×	✓	✓	✓
nrop [38]		✓	✓	×	×	×	×	✓	✓	✓
Q [30]		✓	✓	✓	×	×	×	✓	✓	✓
ROPC [24]		✓	✓	✓	✓	✓	✓	✓	✓	✓
DEPLib [35]		✓	✓	✓	×	✓	✓	✓	×	×
Agafi 1.1 [16]		✓	×	✓	×	✓	✓	✓	×	×
mona.py (rev566) [11]	2.0	✓	×	✓	×	✓	✓	✓	×	×
ROPgadget 5.4 [27]		✓	×	✓	×	✓	✓	✓	✓	✓
rp++ 0.4 [36]		✓	×	×	×	✓	✓	✓	✓	✓
Ropeme [10]		✓	×	×	×	✓	✓	×	✓	×
ropper 1.8.7 [28]		✓	×	✓	×	✓	✓	✓	✓	✓
MSFrop [20]		✓	×	×	×	✓	✓	✓	✓	×

Table 1: Summary of ROP tools. Note that many tools have limitations, bugs, or do not work as expected, which we discuss in § 4.1 and § 5.

ROPgadget lists duplicates, i.e., the same gadget at the same address is listed more than once. We informed the developer about this bug. ROPgadget does not have an option to define the maximum number of instructions in a gadget. Only the maximum number of byte per gadget can be set. We ran our experiments using 110 bytes for the maximum length, leading to an average opcode size of about 3 bytes per instruction. Originally, we planned to use a much larger number to make sure we do not miss any gadgets. However, even with a depth of 110 bytes the evaluation of ROPgadget on Chromium took over 6 hours, consuming 160 GB of RAM. Afterwards, we used a script to go through the results and remove any gadgets that contained more than 35 instructions. Therefore, we miss gadgets that contain 35 or fewer instructions but are longer than 110 byte.
rp++ originally comes with a fixed maximum gadget length of 20 instructions. We modified the source code, changing this upper limit to 35 and recompiled it, so it correctly discovers longer gadgets, too.
ropper ropper does not find some simple and short gadgets and keeps gadgets that contain conditional jumps. Such gadgets are difficult to use, especially since no information is given about which paths are taken under which circumstances.

Since all four tools use slightly different filters or sometimes contain bugs, it is difficult to compare their results. For example, ROPgadget and rp++ keep gadgets that contain privileged instructions (e.g., `in`, `out`, or `hlt`), which terminate the process. ROPgadget's output contains duplicate gadgets, and ropper keeps gadgets that contain conditional jumps, which the other tools do not. We filter and clean the output of all tools, removing any duplicates and privileged instructions as well as jumps. As Table 2 shows, eventually all tools find a similar number of gadgets.

Binary	PSHAPE	rp++	ropper	ROPgadget
firefox_W	6,709	6,182	5,445	6,259
iexplore_W	928	888	836	888
chrome_W	64,372	58,890	52,991	59,969
mshtml_W	1,329,705	1,239,403	1,099,466	1,242,616
jfxwebkit_W	1,172,718	1,076,350	960,091	1,086,061
chromium_L	5,358,283	5,159,712	4,579,388	5,130,856
apache2_L	24,164	22,722	18,061	22,875
openssl_L	6,978	6,829	5,377	6,845
nginx_L	26,314	25,700	21,081	25,245

(a) Number of extracted gadgets

Function	PSHAPE	ropper	ROPgadget
$\text{W}_{VirtualProtect}$	2/4	-	-
$\text{L}_{mprotect}$	4/4	1/4	1/4
L_{mmap}	3/3	-	-

(b) Number of gadget chains

Table 2: (a) Number of gadgets found by each tool on the given binaries, as determined by our evaluation. (b) It is possible to build chains to `mprotect` for all four Linux binaries, line mprotect shows how many of those chains each tool creates. For `mmap`, only three of the Linux binaries have the necessary gadgets to build a chain and this line shows how many of those each tool can create. Chains to `VirtualProtect` exist in four out of the five Windows binaries, this line shows how many of them each tool creates. A dash indicates that the tool does not support calling a function that requires the tool to initialize the required number of arguments. In (a) and (b), L denotes Linux and W, Windows.

Gadget Chaining Here, we evaluate the tools in regards to their ability to create gadget chains. The minimal requirement for a tool to be considered in the experiments is that it can build ROP chains for 64-bit Windows or 64-bit Linux, correctly initializing the registers used for passing parameters to functions. Since most 32-bit calling conventions pass parameters on the stack, ROP chains have to be constructed differently, making a comparison difficult. We use functions that are regularly used in ROP exploits. For Linux, the goal is to create two chains, one that loads registers with analyst-controlled data for invoking a function that takes three arguments (e.g., `mprotect` or `execve`) and one chain that loads registers with analyst-controlled data for invoking a function that takes six arguments (e.g., `mmap`). For Windows, the goal is to create a chain that loads registers with user-controlled data for invoking a function that takes four arguments (e.g., `VirtualProtect` or `VirtualAlloc`). From this point on, we refer to these goals by the function's names but keep in mind that any function using the same number of parameters or fewer can be invoked, too.

From the list of available tools, only ROPgadget and ropper satisfy our requirements. The results of the experiments have been summarized in Table 2b. **ROPgadget** cannot create chains for Windows, does not offer any targets for a ROP chain and instead always tries to build a chain to create a shell using `execve`. However, this function requires initializing three arguments, allowing us

to evaluate at least one goal for Linux. ROPgadget successfully created a chain for chromium, but it did not succeed on any of the remaining binaries.
ropper cannot create chains for 64-bit Windows, but offers two targets for ROP chain creation on 64-bit Linux, `mprotect` and `execve`, which both take three arguments. Again, this allowed us to evaluate at least one of the goals we specified previously. However, for openssl and nginx, ropper was able to initialize only `rdi`, despite discovering several useful and simple gadgets that load the other registers. For apache2, ropper successfully initialized `rdi` and `rdx`. Ropper successfully created a ROP chain for chromium, initializing all three registers used for passing parameters to `mprotect` or `execve`. All gadgets used in the chains are without side-effects and without preconditions. Thus, no additional work to satisfy preconditions is necessary.
PSHAPE successfully created fully functional chains for both `mprotect` and `mmap` for the following Linux binaries: chromium, apache2, and nginx. We present and discuss the chains for apache2 and nginx in § 4.2. For openssl it was only possible to create a chain to `mprotect`. This was due to the fact that no gadget was found to initialize `r9`, which we confirmed manually using both PSHAPE and ROPgadget. On Windows binaries, PSHAPE failed to build chains for firefox.exe and iexplore.exe, and we confirmed, again using both PSHAPE and ROPgadget, that, in fact, the necessary gadgets are not present in the respective binaries. For mshtml.dll and jfxwebkit.dll, PSHAPE successfully built a chain. It also created a chain for chrome.exe, however, it required another gadget to be prepended manually. Hence, we did not count it towards successful chain creations in Table 2b. We discuss this chain and its shortcomings in § 4.2.

In cases where PSHAPE failed to build a chain, we evaluated whether a human analyst could succeed. In other words, we assessed if it was in fact not possible to build a chain, due to a lack of useful gadgets, or if our tool's limitations (see § 6) were to blame. In the case of openssl and iexplore.exe, the former is the case. While there are gadgets that initialize the registers, they are often initialized to a constant value. Other times we found a gadget that does initialize a register to an analyst-controlled value, however, unless that value is a specific constant, a jump is taken in the same gadget, effectively forcing the analyst to initialize the register with that specific value. For firefox.exe, an analyst can create a ROP chain. The gadgets that have to be used are complex, requiring initialization of several gadgets and memory locations to ensure that jumps are not taken. Since PSHAPE is unable to utilize such gadgets, it was unable to automatically generate a chain in this case.

4.2 PSHAPE in Practice

Next we qualitatively evaluate three automatically created chains. Note that any padding required between gadgets is added automatically but omitted here to increase readability and due to size constraints.
Chain for apache2. The chain is presented in Fig. 3a. Gadgets 2 to 7 are used to initialize the registers used for passing parameters. PSHAPE detects that `rax` is dereferenced by gadget 6 and before that, aliased with `ebp` (gadget 4).

Therefore, another gadget is added that initializes `rbp`. An even shorter chain could have been created by arranging the gadgets in such a way, that gadgets 7 and 4 execute before gadget 6. In this case, gadget 7 initializes `rbp`, gadget 4 copies it to `rax`, which is then dereferenced by gadget 6. This would make the first gadget unnecessary. However, PSHAPE does not detect that, as it uses the first permutation whose postconditions are correct (see § 3.2).

Chain for nginx. The chain is presented in Fig. 3b. In the first iteration, the chain consists of gadgets 3 to 8, which are used to initialize the registers used for passing parameters. Gadget 6 dereferences `rax` and `rbx`, which is why PSHAPE initializes these two registers by adding gadgets 1 and 2 to the chain. Gadget 8 dereferences `rbx`, which is initialized by gadget 6.

Chain for chrome.exe. The chain is presented in Fig. 3c. Gadgets 2 to 5 are used to initialize the registers used for passing parameters. PSHAPE correctly detected that there are no better-suited gadgets for initializing `r9` and resorts to using gadget 2, prepended by gadget 1 to make `r15` analyst-controlled. Unfortunately, PSHAPE cannot automatically satisfy the precondition of the `cmovns` instruction, because this conditional `mov` instruction checks the `sign` flag, and currently, PSHAPE ignores flags (see § 6). Therefore, to make sure the chain executes correctly, the analyst has to prepend, e.g., a simple `xor rax, rax` instruction to the chain.

5 Related Work

Here we discuss related work that was not yet covered in § 4.1.

Q [30] takes an existing exploit which does not bypass DEP or ASLR, and attempts to harden it, i.e., rewrite it so it bypasses these mitigation techniques. To bypass ASLR it relies on unrandomized code sections and then uses gadgets from those sections to construct a ROP payload to bypass DEP. The payload is written by the attacker using QooL, Q's own exploit language. In their evaluation, the authors show how Q hardens nine simple stack buffer overflow exploits for Windows and Linux, with a payload that invokes a linked function or `system`/`WinExec`. Q cannot handle gadgets containing pointer dereferences, which our approach not only handles, but also ensures they are safe to use.

ROPC [24] is based on Q, but publicly available. Its main feature is a gadget compiler which takes an input binary and a program written in their own language called ROPL. Then, ROPC creates this program using only gadgets from the input binary. While it looks favourable to our tool on paper, because it is Turing-complete, only a proof of concept prototype, dating back to June 2013, is available. This prototype only works on one included, synthetic example, but did not succeed on the real binaries we use in the evaluation.

BARFgadgets [37] is based on Q and its main focus is classifying and verifying gadgets into various types such as *load register* or *store memory*. It provides very basic summaries that only contain what the first instruction of a gadget does, and which other registers are clobbered.

1	0x3ebe8	**pop** rbp ; **ret** ;
2	0x46774	**pop** rdi ; **ret** ;
3	0x57abd	**pop** rsi ; **ret** ;
4	0x7800d	**pop** rcx ; **mov** eax, ebp ; **add** rsp, 8 ; **pop** rbx ; **pop** rbp ; **ret** ;
5	0x41200	**pop** rdx ; **pop** rbx ; **ret** ;
6	0x4d552	**pop** r8 ; **mov** rax, **qword ptr** [rax] ; **ret** ;
7	0x7800c	**pop** r9 ; **mov** eax, ebp ; **add** rsp, 8 ; **pop** rbx ; **pop** rbp ; **ret** ;

(a) Gadget Chain for apache2

1	0x412dab	**pop** rax ; **add** rsp, 8 ; **ret** ;
2	0x45d594	**pop** rbx ; **ret** ;
3	0x406c20	**pop** rdi ; **ret** ;
4	0x42892b	**pop** rsi ; **ret** ;
5	0x425242	**pop** rcx ; **ret** ;
6	0x444965	**pop** r8 ; **mov qword ptr** [rax], rbx ; **mov** rax, **qword ptr** [rsp + 8] ; **mov qword ptr** [rbx + 0x28], rax ; **mov** rax, **qword ptr** [rsp + 0x18] ; **mov qword ptr** [rbx + 0x18], rax ; **mov edx**, 0 ; **mov** rax, rdx ; **add** rsp, 0x58 ; **pop** rbx ; **pop** rbp ; **pop** r12 ; **pop** r13 ; **pop** r14 ; **pop** r15 ; **ret** ;
7	0x45a8c4	**pop** rdx ; **ret** ;
8	0x424219	**mov** r9, **qword ptr** [rsp + 0x28] ; **mov qword ptr** [rbx + 0x48], r9 ; **mov** r10, **qword ptr** [rsp + 0x30] ; **mov qword ptr** [rbx + 0x50], r10 ; **mov** r11, **qword ptr** [rsp + 0x38] ; **mov qword ptr** [rbx + 0x58], r11 ; **add** rsp, 0x48 ; **pop** rbx ; **pop** rbp ; **pop** r12 ; **pop** r13 ; **pop** r14 ; **pop** r15 ; **ret** ;

(b) Gadget Chain for nginx

1	0x56c63	**pop** r15 ; **ret** ;
2	0x25272	**cmovns** r9d, **dword ptr** [r15] ; **ret** 0x2b48 ;
3	0x9fec6	**pop** r8 ; **ret** ;
4	0x385da	**pop** rdx ; **ret** ;
5	0xa15d3	**pop** rcx ; **ret** 0x6e9 ;

(c) Gadget Chain for chrome.exe

Fig. 3: Three chains created by PSHAPE

nrop [38] finds semantically equivalent gadgets to a given instruction. Our tool could be used for a similar purpose, as semantically equivalent gadgets have the same summary. Until early 2016, nrop's website was online and stated that automatic gadget chaining would be coming soon, however, the tool has not received any updates since then.

ROPER [15] is currently in the early stages of development and will use a genetic component: after gadgets have been found, they will be put together randomly. From this pool, four chains will be selected, executed, their fitness assessed, the two least fit chains killed, the other two chains mated and their children will be added back to the pool. This process will be repeated until it converges on a set of viable chains.

6 Limitations and Future Work

Our prototype implementation currently cannot summarize gadgets that include instructions that check CPU flags (e.g., `cmov`) and filters out gadgets that contain instructions changing the program flow (e.g., `jne`). We plan do address this in future work, as it will enable PSHAPE to successfully build chains for more binaries. In our evaluation it was in all but one cases possible to build a ROP chain without having to incorporate gadgets that contain jumps, however, with mitigation techniques that drastically reduce the number of available gadgets (e.g., Control-Flow Integrity), it will be important to utilize all available gadgets.

Further optimizations are possible, e.g., when combining gadgets we can continue to check for a permutation that has fewer preconditions instead of taking the first permutation that has the correct postconditions.

We plan to add features which help the analyst to find gadgets that are useful for bypassing certain mitigation techniques. E.g., we consider adding a filter to use only call-preceded gadgets, which helps bypass some CFI solutions [25, 39].

7 Conclusion

ROP is the cornerstone of today's low-level exploits, yet tool-support is lacking. Current ROP chain creation requires significant manual work. Here we present PSHAPE, a tool that supports analysts during exploit development. It offers gadget summaries, a compact representation of the effects a gadget has on registers and memory. Furthermore, it automates gadget chaining, loading registers used for passing parameters with analyst-controlled data, and making sure that any preconditions are satisfied.

We compare PSHAPE to twelve other tools in terms of their gadget finding and autochaining abilities. Most of those tools, however, do not work properly in realistic scenarios, contain bugs, or are not available. This left us with three tools to compare to empirically. We applied these tools and PSHAPE to nine widely used binaries, a total of 147 MiB of code, and eleven realistic exploit scenarios. Our tool is the only one that successfully creates ROP chains fully-automated and succeeds in nine out of eleven scenarios. Other tools only create a chain for a single scenario, showing that there is a big gap between the theoretical capabilities of current state of the art tools and their usefulness in practice.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback and suggestions on improving the paper. This work was supported, in part, by NSF CNS-1513783, by the German Federal Ministry of Education and Research (BMBF) and by the Hessian Ministry of Science and the Arts within CRISP (`www.crisp-da.de`), as well as by the Heinz Nixdorf Foundation.

References

1. Andersen, S., and Abella, V. Memory protection technologies. https://technet.microsoft.com/en-us/library/bb457155.aspx, Aug. 2004.
2. Athanasakis, M., Athanasopoulos, E., Polychronakis, M., Portokalidis, G., and Ioannidis, S. The devil is in the constants: Bypassing defenses in browser JIT engines. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014* (2015).
3. Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., and Boneh, D. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 227–242.
4. Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented programming: a new class of code-reuse attack. In *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 30–40.
5. Bray, B. Compiler security checks in depth. http://msdn.microsoft.com/en-us/library/aa290051(v=vs.71).aspx, Feb. 2002.
6. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proc. of the 17th ACM Conf. on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 559–572.
7. Chen, X. Aslr bypass apocalypse in recent zero-day exploits. https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html.
8. Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. Ropecker: A generic and practical approach for defending against ROP attacks. In *NDSS* (2014).
9. Davi, L., Sadeghi, A.-R., and Winandy, M. Ropdefender: a detection tool to defend against return-oriented programming attacks. In *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2011), ASIACCS '11, ACM, pp. 40–51.
10. Dinh, L. L. Ropeme - rop exploit made easy. https://github.com/packz/ropeme.
11. Eeckhoutte, P. V. mona.py. https://github.com/corelan/mona.
12. Federico, A. D., Cama, A., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. How the elf ruined christmas. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), USENIX Association, pp. 643–658.
13. Follner, A., Bartel, A., and Bodden, E. Analyzing the gadgets - towards a metric to measure gadget quality. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings* (2016), pp. 155–172.
14. Follner, A., and Bodden, E. Ropocop - dynamic mitigation of code-reuse attacks. *Journal of Information Security and Applications* (2016), –.
15. Fraser, L. Roper. https://github.com/oblivia-simplex/roper.
16. Gallo, M. Agafi. https://github.com/CoreSecurity/Agafi.
17. Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 575–589.
18. Howard, M., Miller, M., Lambert, J., and Thomlinson, M. Windows isv software security defenses. http://msdn.microsoft.com/en-us/library/bb430720.aspx, Dec. 2010.

19. Hund, R., Willems, C., and Holz, T. Practical timing side channel attacks against kernel space aslr. In *Proc. of the 2013 IEEE Symp. on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 191–205.
20. Metasploit. Msfrop. `http://www.offensive-security.com/metasploit-unleashed/msfrop`.
21. Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices* (2007), ACM.
22. Nguyen, A. Q. Capstone: Next generation disassembly framework. `http://www.capstone-engine.org/BHUSA2014-capstone.pdf`.
23. Nguyen, A. Q. Optirop. `https://media.blackhat.com/us-13/US-13-Quynh-OptiROP-Hunting-for-ROP-Gadgets-in-Style-WP.pdf`.
24. Pakt. Ropc. `https://github.com/pakt/ropc`.
25. Pappas, V., Polychronakis, M., and Keromytis, A. D. Transparent rop exploit mitigation using indirect branch tracing. In *Proc. of the 22Nd USENIX Conf. on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX, pp. 447–462.
26. Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. 15*, 1 (Mar. 2012), 2:1–2:34.
27. Salwan, J. Ropgadget. `https://github.com/JonathanSalwan/ROPgadget`.
28. Schirra, S. Ropper - rop gadget finder and binary information tool. `https://scoding.de/ropper/`.
29. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.-R., and Holz, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *36th IEEE Symposium on Security and Privacy (Oakland)* (2015).
30. Schwartz, E. J., Avgerinos, T., and Brumley, D. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 25–25.
31. Serna, F. J. The info leak era of software exploitation. `http://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf`, 2012.
32. Shacham, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM Conf. on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM.
33. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM Conf. on Computer and communications security* (2004), CCS '04, ACM.
34. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS* (2015).
35. Sol, P. Deplib. `https://www.immunitysec.com/downloads/DEPLIB.pdf`.
36. Souchet, A. rp++. `https://github.com/0vercl0k/rp`.
37. STIC, P. Barfgadgets. `https://github.com/programa-stic/barf-project`.
38. Wailly, A. nrop. `https://github.com/awailly/nrop`.
39. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity and randomization for binary executables. In *Proc. of the 2013 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2013), SP '13, IEEE Computer Society, pp. 559–573.