Static Analysis of ROP Code

Daniele Cono D'Elia, Emilio Coppa, Andrea Salvati, Camil Demetrescu
Sapienza University of Rome
{delia,coppa,demetres}@diag.uniroma1.it
andrea.slvt94@gmail.com

ABSTRACT

Recent years have witnessed code reuse techniques being employed to craft entire programs such as Jekyll apps, malware droppers, and persistent data-only rootkits. The increased complexity observed in such payloads calls for specific techniques and tools that can help in their analysis. In this paper we propose novel ideas for static analysis of ROP code and apply them to study prominent payloads targeting the Windows platform. Unlike state-of-the-art approaches, we do not require the ROP activation context be reproduced for the analysis. We then propose a guessing mechanism to identify gadget sources for payloads found in documents or over the network.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; *Malware and its mitigation.*

KEYWORDS

Return oriented programming, code reuse, static analysis, exploits.

ACM Reference Format:
Daniele Cono D'Elia, Emilio Coppa, Andrea Salvati, Camil Demetrescu. 2019. Static Analysis of ROP Code. In *12th European Workshop on Systems Security (EuroSec '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3301417.3312494

1 INTRODUCTION

Code reuse techniques have been historically used to circumvent system-level defenses such as Data Execution Prevention, borrowing fragments from the program's code or a library linked to it to perform remote code execution. Return Oriented Programming (ROP) [11] is the most prominent instance of such mechanisms.

ROP is popularly known as an exploitation technique to get around executable-space protection mechanisms and perform shellcode injection. However, recent years have witnessed uses of ROP in increasingly more complex scenarios. For instance, researchers have employed ROP to get around code review and signing mechanisms in iOS, building apparently innocent Jekyll apps [13] that turn evil at a remote attacker's command and carry out unforeseeable malicious actions. The weaponization of CVE-2013-0641 for Adobe Reader is the first embodiment of a ROP-only attack in a document, with a chain of unprecedented length and complexity embedded in a PDF to bypass Adobe Sandbox without using shellcode. ROPInjector [10] rewrites instead portions of a malware as ROP chains for the sake of anti-virus evasion via polymorphism. Finally, the speculation of return-oriented rootkits to bypass kernel integrity mechanisms has become reality with Chuck [12], the first persistent data-only malware in the form of a Linux rootkit.

The increase in complexity of ROP payloads calls for tools and systems that can assist humans in the analysis of such sequences, provided that existing reverse engineering approaches and resources tailored to EIP-driven control flow are not well-suited for this goal. To the best of our knowledge, only two works have dealt with this problem. deRop [9] attempts to convert a ROP exploit to a semantically equivalent shellcode that can be analyzed by existing technologies. However, the implementation was tested against very simple, "classic" exploits and not released. ROPMEMU [7] uses emulation over a memory dump of the system, collecting, simplifying and merging multiple traces obtained via a coarse-grained multi-path exploration of the ROP chains of the Chuck malware. The work sheds light on the main challenges in the analysis of complex chains, and its implementation is available to the community.

Contributions. In this work we explore the analysis of ROP code from a new standpoint: rather than converting ROP sequences to simplified EIP-based representations, we identify, dissect, and annotate code components (program points, basic blocks, branching sequences, and API calls) at the ROP level, providing the analyst with an overview of the inner workings of a ROP program. Other than favoring code understanding, we believe the proposed shift may enable new applications like retrofitting and similarity analysis for exploits, and aid the analysis of programs obfuscated with ROP.

Our technique can be used without having to reproduce the ROP activation context (e.g., an exploitation attack, or in the Jekyll case a remote interaction), as only the gadget sources must be known. We also propose algorithms to identify such sources for payloads spotted in malicious documents or over the network.

We test our ideas on ROP payloads written for Windows, and make a prototype implementation available to the community.

2 APPROACH

We now provide the reader with a high-level overview of our approach, followed by an in-depth discussion of each component of the analysis. We seek for a technique that can be applied statically, without knowing the memory context and activation sequence for a chain or relying like ROPMEMU on a memory dump, possibly taken in the exact moment execution first jumps into ROP code—as a gadget may break the chain by polluting the stack once executed.

Overview. Given a ROP program and one or more PE files (e.g., main executable, DLLs) containing the borrowed gadgets, we use

EuroSec '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6274-0/19/03…$15.00
https://doi.org/10.1145/3301417.3312494

emulation and multi-path exploration to uncover instructions and control flows the code may exercise. A loader component populates the emulator's memory with the ROP payload and the executable sections from the PE files. Memory accesses to non-code regions are trapped and handled to attempt to fill gaps with a dynamic analysis in the exploited run-time context. Shadow copies of memory and stack are maintained to keep track of changes and their propagation. Special care is taken to identify function calls to standard Windows components and control transfers to non-ROP code or to dynamically generated chains (§2.1). Program points, basic blocks, and branching machinery are identified at ROP level rather than in a standard EIP-oriented sense, while gadgets within blocks are annotated with tags obtained, e.g., using a constraint solver (§2.2).

We then discuss the analysis of payloads captured over the network or in documents, as determining the victim application and/or the gadget source may not be immediate in such scenarios. Leveraging details of the ASLR mechanism of Windows, we propose a strategy (§2.3) for guessing PE components required by chains that are generated at run time (e.g., using information from a leaked pointer) or use hard-wired address (e.g., as in the Jekyll approach).

2.1 Emulation

We implement our ideas in Unicorn, a CPU emulator framework that supports fine-grained instrumentation on different architectures. Our prototype supports 32-bit Windows PE executables and libraries, and we are planning to extend it to Linux ELF files.

We assume the analyst has got access to a ROP payload—captured for instance by analyzing network communications or documents, or spotted in the wild by other actors—and that knows which program will host it, including the component(s) from which the gadgets are borrowed. In §2.3 we will relax the latter assumption.

Execution Context. We set up the emulator by loading the ROP payload to an arbitrary memory area, with the stack pointer ESP pointing to the initial part of the chain and the instruction pointer EIP to the first gadget in it. We load code sections and headers for each PE component referenced in gadget addresses, choosing the right image base address to materialize them. Note also that when gadgets are borrowed only from shared libraries we do not need to load the PE of the host application, unless its Import Address Table (IAT) is used by ROP code to locate API functions.

As the ROP program would run in an unnatural, skimmed address space, we need a mechanism to identify memory accesses that are legit in the context of a real execution and possibly mimic pre-existing contents that are manipulated by the program[1]. We employ a shadow memory that tracks every time an address within an initially unmapped page is accessed in a read or write operation. We create a valid page using a page fault handler on the first access to a region, filling its content with a marker byte, and set up a bitmap to keep track of addresses accessed in it, as well as of the ones seeing repeated read/write operations. In ROP code it is common to observe irrelevant memory operations as side effects of gadgets alongside those required by the payload's logic, such as to transfer data to a region just allocated by it. Maintaining a shadow memory lets us track effects and prerequisites on memory for each step of the execution, and is useful also for multi-path exploration.

We monitor execution conditions for which the instruction pointer ends up in an executable region either allocated by the payload or for which it modified the permissions (as often is the case with the stack) to account for possible control transfers to injected code sequences. Conditions are dynamically updated as the chain executes operations such as VirtualAlloc or VirtualProtect.

Function Calls. The usage of system library and system calls in ROP code is nowadays not limited anymore to making a region executable and copying a shellcode to it. For instance, the weaponization of CVE-2013-0641 creates and decrypts a DLL file on disk to later load it into the host process. The authors of ROPMEMU remark that function calls need special care as they usually take place with the same control transfer mechanism used for gadget chaining, i.e., as a return to the function's entry point. They use the distance from the first return instruction to discriminate between gadgets and function bodies, and produce random return values for functions with the exception of, e.g., those for allocating memory.

In our skimmed address space the invocation of functions that belong to a loaded PE component are detected by monitoring jumps to their prologue (we precompute function entrypoint addresses using Nucleus [2]). However, API functions often belong to PE components (such as Windows libraries or the Visual Studio runtime) that may differ from the ones used as gadget sources. We devise a strategy that can help identifying missing components of the address space—which would be present instead in a dump—and steer the emulation process when a call takes place:

- for functions accessed by dereferencing the IAT of a loaded PE image, we parse the IAT at load time to identify for each entry the DLL and the referenced function symbol (export) within it, and rewire the target address to a non-executable memory area, so that we can hook such calls using the handler for fetch protection exceptions in the emulator;
- for function accessed via hard-wired addresses[2], we rely on a database of pre-computed offsets for exports of essential Windows DLLs such as kernel32.dll, and check for matches for the combination of export offset and valid base address;
- we complement the previous technique with a heuristic that inspects the calling context and tries to guess the most likely candidate based on the arguments and their order: for instance, immediate 0x40 (EXECUTE_READWRITE) is typically passed as third argument to VirtualAlloc and as fourth to VirtualProtect in many ROP payloads;
- when the previous techniques fail or multiple guesses are available, we present the analyst with some options, including the generation of a random return value for the call as in the default strategy of ROPMEMU.

To handle direct invocations of system calls and exotic mechanisms typical of ROP code, such as writing the ordinal to a register and jumping halfway in Nt library wrapper of another syscall, we use a database of system call ordinals for different Windows version.

When a call to a commonly used function is intercepted, we rely on models implemented as simple Unicorn plugins written

[1] For instance, API addresses that the Windows loader solves and writes in the IAT. As a direction for future work, we plan to explore symbolic execution techniques (e.g., [3, 5]) to account for other possibly unknown dependencies in the host program.
[2] This accounts for non-randomized DLLs, functions solved by the host via GetProcAddress, and chains generated by an active remote attacker as in the Jekyll architecture.

in Python to mimic the effects of the call in the context of the emulator. In the implementation we currently support functions for string manipulation, memory allocation, file handling, dynamic function/DLL loading, and simple decompression schemes.

2.2 Code Analysis

Branch Identification. Reconstructing the control flow of ROP code can be done by unraveling the chain and monitoring how ESP varies. In particular, fall-through edges between gadgets correspond to stack pointer modifications of a byte amount equal to the *memory footprint* of the source gadget, represented by its operands (i.e., values popped from the stack), any present padding (e.g., for `ret N` instructions), and by its own address. Branches instead are normally implemented using gadgets that before reaching the return instruction add to ESP a quantity (variable in conditional branches, and fixed in goto-like jumps) typically read from a register.

Conditional branches are not straightforward to implement in ROP code, as classic conditional jump instructions do not mix well with return-oriented control transfers. For this reason, such a branch in ROP is typically implemented in three steps:

(1) an arithmetic or logic instruction sets one or more flags in the status register EFLAGS to reflect the semantics of the required check (e.g., a gadget that executes `neg eax` sets the carry flag CF to 1 if EAX contains 0, and to 0 otherwise);
(2) an instruction leaks one or more bits of EFLAGS to a register or memory, having its value reflect the outcome of the comparison (e.g., for ECX initially set to 0, a gadget that executes `adc ecx,ecx` will modify ECX to 1 if and only if CF=1);
(3) the value from the previous step is used to compute an offset that in turn is added to ESP. Having 0 as resulting offset implies that the branch is not taken.

We thus devise the following strategy to identify the components of a ROP branch. For each condition flag we keep track of the last instruction (and relative gadget) that modified it; this information can be obtained from a disassembler like Capstone. We then monitor instructions that can leak one or more condition flags to registers or memory (e.g., `adc`, `pushf`, `lahf`), and for each flag we perform a simple data flow analysis to determine which gadgets manipulate it, following taint propagation for both registers and memory till the instruction that adds the tainted computed offset to ESP is reached.

Multi-path Exploration. Branch identification is valuable for control flow reconstruction, but in our setting also a preparatory step for the multi-path exploration that we use to identify possible alternative control flows in a ROP program. To support this strategy, every time we encounter an instruction that leaks condition flags we create a snapshot of the shadow memory and of the CPU context. When the current control flow path reaches its end, we restart the emulator from the snapshot and flip the condition flag(s) that taint analysis identified as involved in the ESP offset computation.

Our multi-path exploration reconstructs an over-approximation of the possible control flows of the program, as some branches may not be feasible in a concrete execution, for instance if the ROP code is obfuscated. This limitation is shared also by ROPMEMU. We improve on the technique proposed by its authors in two ways: we propose a simple scheme to identify the three components of branches[3] in ROP code, and we use taint analysis to rule out repeated paths from the analysis (i.e., we flip only flags actually involved in a branch) which could arise from unintended side effects or adversarial obfuscation strategies.

We plan to refine our scheme by adding symbolic execution to the picture: we believe it could not only help ruling out some unfeasible branches, but also provide a logical representation of the computed conditions via the constraints collected in the exploration.

Control Flow Reconstruction. Due to the return-oriented control transfer mechanism, standard control flow graph (CFG) reconstruction algorithms cannot be applied directly to ROP code.

For the notion of program point, which in compiler-generated code corresponds to instruction locations, we can use the distinct <ESP value, gadget address> pairs that the ROP payload can yield. This choice accounts for having the same gadget used in multiple points of the chain, and for cases where the same memory location may see different contents as in, e.g., dynamically generated chains or ROP unpacking mechanisms. With respect to ROP branches, they simply are indirect branches, and multi-path emulation can be used to reveal the targets of conditional ROP jumps.

As the emulation proceeds, we build a directed graph where nodes represent ROP program points and edges indicate control transfers (a fall-through or a jump sequence) between them. Transfers to non-ROP code (library and system calls) are marked as nodes of interest in the graph. The graph construction process accommodates for multiple explored execution paths, as the information on the current node (i.e., program point) is stored in the state snapshot used by multi-path analysis, and new edges and nodes can be added as each path gets explored. Once code analysis is complete, we merge nodes that form sequences with single points of entry and exit, thus assembling the ROP basic blocks of the CFG.

Annotations. Our code analysis process pursues different goals compared to deRop and ROPMEMU, which try to convert ROP code to a simplified EIP-based representation. The techniques and transformations used in these works could still be used in our setting, for instance to provide an additional simplified representation of the instructions in a ROP basic block.

To assist analysts as they weasel their way through ROP code, we augment the CFG representation with annotations on the program semantics carried out at each step. For instance, we identify patterns of gadgets within ROP basic blocks, and represent them as a whole inside them. We already mentioned the highlighting of transfers to non-ROP code, and how we identify the components in a branch decision. We also leverage an existing analyzer component from the BARF suite to extract—using a combination of code lifting and verification in a constraint solver—a semantic signature of each gadget, that is, an assembly-like description of the operation carried out by it according to the operation category (e.g., memory load/store, register assignment, arithmetic) it most likely falls into.

While in our analysis pipeline this is probably the less mature component, we believe an interesting research problem lies behind it. To the best of our knowledge, gadget analysis has largely been explored from the attacker's perspective (i.e., finding the right gadget for an operation), but never from a code understanding one.

[3]The authors of ROPMEMU focus on `pushf` instructions and flip the ZF flag, as it is the sole mechanism used to implement branches in the Chuck rootkit they analyze.

2.3 Gadget Guessing

In §2.1 we assumed that the analyst knows the application component(s) from which the payload draws the gadgets. We now try to relax this assumption, proposing an algorithm to identify candidate code modules from a collection of libraries and executables from different programs. We believe this algorithm can be useful in at least two scenarios: one where a ROP payload is captured from the network traffic of an organization [8], for instance as part of a Jekyll app-based attack, and one where static inspection of a document reveals the presence of a ROP chain, but the targeted viewer application and/or its vulnerable version(s) are unknown.

The algorithm leverages both features of Address Space Layout Randomization (ASLR) implementations in mainstream systems and distinctive features of the ROP control transfer mechanism.

Popular operating systems randomize only the base address of a code module, leaving the relative distances between gadgets in a component unaltered. For Windows, only 256 addresses can be used as base for the main image, while libraries are placed using a bitmap of available memory locations within a designated area. The image of an application is divided in consecutive *pages* of 64K bytes, and the first page is loaded at a randomized base address.

Observe also that in the absence of ROP branches or function calls, control transfers in a ROP payload are determined by `ret <N>` instructions at the end of each gadget. However, both are unlikely to be found at the very beginning of a ROP payload.

Explore. We can leverage the above observations to build Algorithm 1. Procedure `explore` simulates the execution of a ROP chain for a fixed number of steps using gadgets from a PE module loaded at some base address. A helper method *getGadgetDelta* fetches the ESP variation yielded by instructions in the gadget used at position *idx* in the chain: intuitively, in a branchless exploration only constant positive offsets are allowed. Note that any δ_{RET} immediate for the `ret` instruction is added to ESP by the CPU after the control transfer takes place, i.e., it affects *idx* at the next step (δ'_{RET} is updated at line 8). When a valid gadget is found for the current choice of (PE, base address) and the resulting displacement for *idx* falls within the bounds of the chain, the exploration can advance by one step. In our experiments, a wrong choice of PE or base was typically revealed in at most 3 steps.

The procedure is used by the two functions shown in Algorithm 2. Depending on the characteristics of the extracted ROP payload, we could be dealing with a chain either made of hard-coded addresses or obtained by adding offsets to an address leaked at run time. In both cases, we wish to identify (PE, base address) candidates that we can fully validate using the emulation approach of §2.1.

Fixed Addresses. The first case accounts for payloads leveraging non-randomized libraries (for instance, as in malicious documents targeting old viewer versions, or in many PoC exploits available online) or generated following the Jekyll approach, where a remote attacker uses a previously leaked pointer to craft and send to the application a payload that is consistent with the randomization choices performed by ASLR. For non-randomized libraries, one could use `explore` over the preferred base address of each DLL to discriminate potential candidates[4]. In the general case, function `guessFixed` exploits ASLR implementation details to guess the base address. As every code page is 64K-byte long, we extract the address of the page in which the first gadget in the chain falls by clearing the 16 LSBs of its address. As code from the considered PE spans *numPages* chunks, we use the gadget page address to explore the possible bases. Line 4 will compute base addresses such that first gadget falls in every possible page from the first to the last. Algorithm 1 will check the validity of subsequent gadgets falling in different pages and rule out malformed alignments (line 4).

procedure `explore`(*PE, chain, idx, depth,* $\delta'_{RET} = 0$)**:**
1 **while** *depth > 0* **do**
2 $G_{\text{ADDR}} \leftarrow$ chain[idx:idx+3]
3 $\Delta_{\text{G-BASE}} \leftarrow G_{\text{ADDR}}$ - PE.base
4 **if** $\Delta_{G\text{-}BASE} < 0$ ***or*** $\Delta_{G\text{-}BASE} >$ *PE.size* **return** false
5 $(\delta_{\text{ESP}}, \delta_{\text{RET}}) \leftarrow$ getGadgetDelta(PE, $\Delta_{\text{G-BASE}}$)
6 idx $\leftarrow$ idx + δ_{ESP} + δ'_{RET}
7 **if** $\delta_{ESP} \leq 0$ ***or*** *invalidIdx(chain, idx)* **return** false
8 depth $\leftarrow$ depth - 1 ; $\delta'_{\text{RET}} \leftarrow \delta_{\text{RET}}$
9 **return** true

Algorithm 1: Exploration simulation for ESP modifications.

initially: results $\leftarrow \langle\rangle$
function `guessFixed`(*PE, chain, startIdx, depth*)**:**
1 numPages $\leftarrow$ (PE.size + 0xFFFE) » 16
2 gadgetPageAddr $\leftarrow$ chain[startIdx:startIdx+3] » 16 « 16
3 **for** *pageIdx* $\in$ *{0 ... numPages-1}* **do**
4 PE.base $\leftarrow$ gadgetPageAddr - (pageIdx » 16)
5 **if** *explore(PE, chain, startIdx, depth, 0)* **then**
6 results $\leftarrow$ results $\cdot \langle (PE, PE.base, 0) \rangle$
function `guessLeak`(*PE, pChain, startIdx, depth*)**:**
7 base $\leftarrow$ *<pick any valid base>*
8 chain $\leftarrow$ applyOffset(pChain, base)
9 **for** *RVA* $\in$ *PE.RVAs* **do**
10 PE.base $\leftarrow$ base - RVA
11 **if** *explore(PE, chain, startIdx, depth, 0)* **then**
12 results $\leftarrow$ results $\cdot \langle (PE, base, RVA) \rangle$

Algorithm 2: Gadget hunting techniques.

Leaked Pointers. The second case accounts for payloads that are dynamically tailored to the runtime by means of a vulnerability that leaks a pointer. This is often the case with documents embedding code written in a scripting language (e.g., JavaScript for PDF documents, or PostScript as in the weaponization of CVE-2015-2545 for Microsoft Office). Such code typically leaks one between the image base, the address of the `.text` segment, and a function pointer, and generates a ROP chain using the leaked pointer to compute gadget addresses. Note that any leaked pointer can be expressed as the sum of the PE base and the RVA (relative virtual address) representing its distance from it. We make use of this observation in the formulation of function `guessLeak`: we choose any valid 64K-aligned address to patch the chain, and attempt an exploration by taking into account every possible RVA. The key to make it work while patching the chain only once is to alter the PE base passed to `explore`: subtracting the RVA from it (line 10) is equivalent to generating[5] a new chain using base+RVA as leaked pointer. RVAs for DLL exports are contained in library headers, while we rely on Nucleus for detecting functions in executables.

[4] From Windows 8 however ASLR may be enforced when relocation data is available.
[5] Algebraic sums are the only operations we witnessed for patching in our experience.

Exploits	Adobe Reader U3D, Audio Converter, BigAnt Server, Castripper, ComSndFTP (2), Easy File Sharing Web Server (3) Free MP3 CD Ripper, Mini-stream RM-MP3 Converter (2), MPlayer, NetTransport, nfsAxe [source: ExploitDB]
Documents	Weaponization of CVE-2013-0641 ("number of the beast")

Table 1: ROP payloads for testing code emulation.

Discussion. Our algorithms hinge on the assumption that a chain in its initial portion does not make non-constant ESP modifications, nor it rewrites itself by pushing addresses of successor gadgets[6]. Candidate PE images are identified using code from Algorithm 2, and false positives are eventually ruled out via full code emulation. A minor refinement we adopt in the implementation is to consider as exploration depth how many distinct gadgets are traversed.

In the presence of more than a few candidates to validate, we would like to integrate gadget quality metrics [6] to prioritize chains that when materialized on a PE image feature higher-quality gadgets, i.e., better suited for writing ROP payloads (for instance, in terms of clobbered registers or unintended memory accesses). When only partial matches are found or full validation reveals later gadget addresses outside the loaded PE, two or more images may be involved[7], and we repeat Algorithm 2 to fill in the blanks.

Observe that Algorithm 1 can run on program points other than the initial. This may turn out useful for instance in exotic scenarios where the stack pivoting hijacks ESP somewhere inside the payload and determining the location of the initial gadget may not be easy. Previous research has used clustering and statistical properties to distinguish between gadget addresses and data/padding. We plan on devising heuristics to identify likely valid program points by inferring possible ESP variations just from the looks of the chain, and refine the choices of idx and δ'_{RET} to be used for `explore` in Algorithm 2 when trying to determine the leveraged PE image.

3 EVALUATION

We report preliminary results from an investigation of our techniques over a collection of ROP payloads for Windows. To facilitate the study of ROP techniques and subsequent research, we make the execution setup available to the community. Our presentation is three-pronged, following the organization of our architecture.

Code Emulation. To see to which extent the execution of ROP code found in the wild can be carried out in our skimmed execution setting outlined in §2.1, we consider 16 variants of exploitation attacks against 12 vulnerable Windows applications, and the ROP-only weaponization of CVE-2013-0641, also dubbed "number of the beast" by the FireEye firm (Table 1). For each payload, we verify that the actions seen in the emulator are consistent with findings from manual analysis and public write-ups when available.

The considered exploits exercise a good deal of Windows APIs, triggering all the function call identification heuristics from §2.1. Invoked APIs involve mainly memory allocation/protection (using standard functions or unusual techniques like creating an empty file and mapping it to later overwritten executable memory) and manipulation (to transfer a shellcode). We check that the emulation halts when about to jump into shellcode, and verify its contents.

The analysis of the "number of the beast" weaponization was significantly more challenging. The payload drops a second-stage DLL and consists of a straight-line sequence of 349 gadgets that:

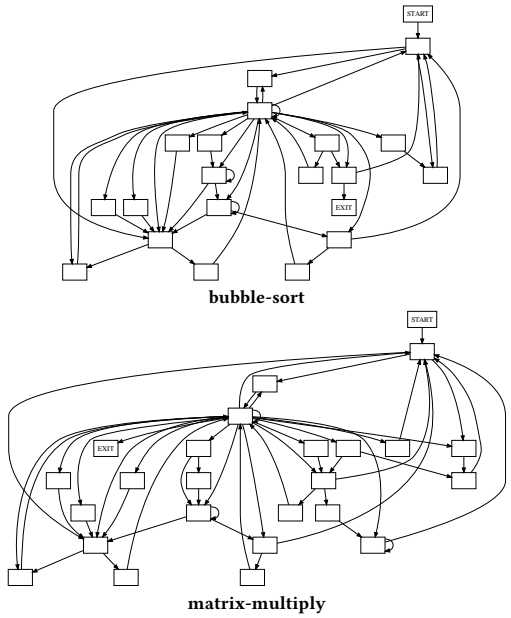

Figure 1: CFG from EIP-based construction algorithm.

- calls LoadLibraryA() on 4 standard Windows DLLs (msvcr90, crypt32, ntdll, kernel32) and solves 5 functions within them throughout the execution using GetProcAddress();
- performs Base64-decoding of a buffer with an API of crypt32 and decompresses it using RtlDecompressBuffer() of ntdll;
- creates a temporary file and writes the DLL payload to it, reading RVAs in msvcr90 for fopen() and fclose() from the IAT and solving fwrite() dynamically with GetProcAddress();
- loads the dropped DLL with LoadLibraryA, then sleeps.

Our emulation mechanism could run the ROP payload in full once we added Python models to mimic the effects of CryptStringToBinaryA (for Base-64 decoding) and RtlDecompressBuffer.

Code Analysis. The branchless nature of payloads from Table 1 was not adequate to test the techniques described in §2.2. We thus discuss the ROP programs analyzed in [14] as showcases of how ROP "leads to program logic that can be tricky to decipher". Unfortunately, we could not analyze portions of the Chuck malware as at the time of writing the repository linked in [12] was not working.

[14] discusses four ROP programs: *bubble-sort*, *factorial*, *fibonacci*, and *matrix-multiply*. For *factorial* and *fibonacci*, the authors show the dynamic CFG obtained with a classic algorithm from the compiler literature [1], where basic block boundaries are determined by targets of control transfer instructions. They then compare CFGs of C programs encoding the same operations of the ROP ones with CFGs obtained after applying their trace simplification technique.

We reconstruct EIP-based dynamic CFGs for the benchmarks, reporting the two most complex cases in Figure 1. Due to lack of space, we refer the reader to [14] for the graphs for *factorial* (18 gadget blocks, 36 edges) and *fibonacci* (18 gadget blocks, 32 edges); the graphs from Figure 1 are not reported in the work.

[6] We are not aware of self-modifying ROP code to date. Yet, it remains a possibility.
[7] When ASLR is enforced, this is difficult: an attacker either needs multiple pointer leaks, or has to discover the address of a module from the image contents of another.

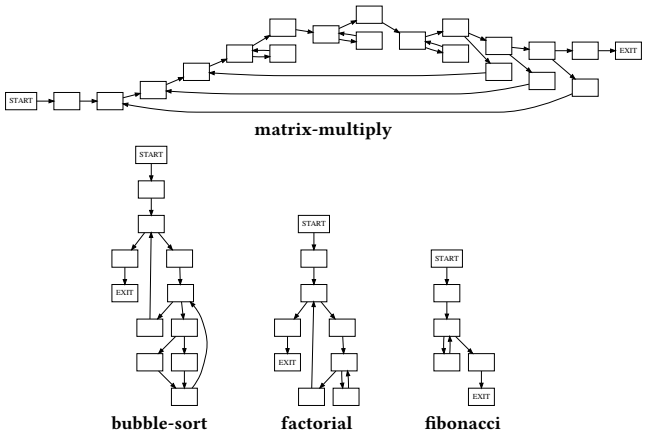

Figure 2: ROP CFG with multi-path exploration.

We then perform multi-path exploration on the programs without providing any input, but relying on our branch identification technique to reveal alternative control paths. We use <ESP value, gadget address> pairs as program points and construct ROP basic blocks accordingly. We present the obtained ROP CFGs in Figure 2.

Our control flow reconstruction process resulted in a ROP CFG identical to the CFG of the C counterpart of *factorial* shown in [14], while for *fibonacci* the graphs are very similar (our clique of two mutually adjacent nodes corresponds to a single block with a self loop in the C CFG). [14] does not discuss *bubble-sort* and *matrix-multiply*, so we inspect them manually. For the former, we identify an outer loop with an inner loop for comparisons and swaps; for the latter, we identify three nested outer loops for indexing, and three tight loops for multiplication implemented with addition gadgets.

Gadget Guessing. We conclude by presenting preliminary results from applying our gadget guessing algorithms to payloads from a collection of malicious PDFs targeting different Adobe Reader vulnerabilities (Table 2). ROP code here is typically laid out in memory by means of a Javascript object embedded in the document. Forensic tools such as *peepdf* can extract and prettify the script, as it may be slightly obfuscated in its syntax. Such a script typically checks the application version using the `app.viewerVersion` variable and tailors the chain generation process accordingly or aborts.

We assemble a collection of candidate DLLs and plug-in files for 9 releases of Reader (8.0, 8.3.0, 9.5.3, 10.0.7, 10.1.2, 10.1.4, 11.0.1, 17.008.30051, 18.01120038). For gadget extraction, we use *ROPgadget* and *ropper* to look for gadgets of at most 10 bytes or 6 instructions (the respective defaults) and yielding a constant positive ESP delta.

The chain generation process varies across the samples we consider. S1 and S2 use hard-wired addresses, which hint at a non-randomized DLL; S3 and S4 generate a chain by summing offsets to a leaked pointer; S5 adds to every gadget in an existing chain the difference between the leaked pointer and a value that turns out to be the base address of the original chain.

The code for chain generation is conceptually simple: its input dependencies may be the viewer version and/or a leaked pointer: in our experience, their uses could easily be spotted in the code without resorting to dataflow analyses. Once we define variables for them, we run the code in isolation in an off-the-shelf Javascript engine (we use V8) and retrieve the generated chain. By assigning a valid base address as pointer value we realize the *applyOffset* subroutine used in `guessLeak`. As for the version, we can look at instructions that check it or try random assignments and compare the output chains. S5 does not check it, so releases for which the payload may likely work are not given away by its code. We have been able to identify vulnerable components from our set of versions correctly, incurring no false positives. We determine the leaked pointer from the leveraged PE to be the address of its `.text` section for S3, and its image base for S4 and S5.

ID	CVE	PE for gadgets (version)	Addresses
S1	2010-2883	icucnv34.dll (8.0, 8.3.0), icucnv36.dll (9.5.3)	fixed
S2	2011-2462	icucnv36.dll (9.5.3)	fixed
S3	2013-0641	AcroForm.api (10.0.7, 10.1.2)	patched
S4	2013-2729	AcroRd.dll (9.5.3)	patched
S5	2018-4990	EScript.api (18.01120038)	patched

Table 2: PDFs and leveraged Adobe Reader components.

For validating `guessFixed` on hard-coded payloads leveraging randomized PEs, we synthesize some by rebasing exploits in Table 1 and chains in S1, S2 and S5, as Jekyll attacks for Windows have been speculated only recently [4]. The algorithm reports the correct PE among the components of the considered applications, while full emulation rules out the (rather few) false positives we find.

4 CONCLUSION

We have presented novel code analysis techniques that can be applied directly to ROP programs, discussing directions for future work throughout the paper. Hoping that other researchers may benefit from them, we make our code available at `https://github.com/season-lab/ropdissector/`. This work is supported in part by a grant of the Italian Presidency of the Council of Ministers.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd ed.).* Addison-Wesley Longman.
[2] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *EuroS&P '17*. IEEE, 177–189.
[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Assisting Malware Analysis with Symbolic Execution: A Case Study. In *CSCML '17*. Springer, 171–188.
[4] Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. The ROP Needle: Hiding Trigger-based Injection Vectors via Code Reuse. In *SAC '19*. ACM, 1962–1970.
[5] Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Rethinking Pointer Reasoning in Symbolic Execution. In *ASE '17*. IEEE, 613–618.
[6] Andreas Follner, Alexandre Bartel, and Eric Bodden. 2016. Analyzing the Gadgets. In *ESSoS '16*. Springer, 155–172.
[7] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. 2016. ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks. In *ASIA CCS '16*. 47–58.
[8] Xusheng Li, Zhisheng Hu, Yiwei Fu, Ping Chen, Minghui Zhu, and Peng Liu. 2018. ROPNN: Detection of ROP Payloads Using Deep Neural Networks. *arXiv*.
[9] Kangjie Lu, Dabi Zou, Weiping Wen, and Debin Gao. 2011. deRop: Removing Return-oriented Programming from Malware. In *ACSAC '11*. ACM, 363–372.
[10] Giorgos Poulios, Christoforos Ntantogian, and Christos Xenakis. 2015. ROPInjector: Using Return Oriented Programming for Polymorphism and Antivirus Evasion. *Black Hat USA* (2015).
[11] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. In *ACM TISSEC*.
[12] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. 2014. Persistent Data-only Malware: Function Hooks without Code. In *NDSS '14*.
[13] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. 2013. Jekyll on iOS: When Benign Apps Become Evil. In *USENIX Security '13*. 559–572.
[14] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *SP '15*. IEEE, 674–691.