Rope: Covert Multi-Process Malware Execution with Return-Oriented Programming

Daniele Cono D'Elia, Lorenzo Invidia, and Leonardo Querzoni

Dept. of Computer, Control, and Management Engineering
Sapienza University of Rome
`delia@diag.uniroma1.it`

Abstract. Distributed execution designs challenge behavioral analyses of anti-malware solutions by spreading seemingly benign chunks of a malicious payload to multiple processes. Researchers have explored methods to chop payloads, spread chunks to victim applications through process injection techniques, and orchestrate the execution. However, these methods can hardly be practical as they exhibit conspicuous features and make use of primitives that anti-malware solutions and operating system mitigations readily detect. In this paper we reason on fundamental requirements and properties for a stealth implementation of distributed malware. We propose a new covert design, Rope, that minimizes its footprint by making use of commodity techniques like transacted files and return-oriented programming for covert communication and payload distribution. We report on how synthetic Rope samples eluded a number of state-of-the-art anti-virus and endpoint security solutions, and bypassed the opt-in mitigations of Windows 10 for hardening applications. We then discuss directions and practical remediations to mitigate such threats.

Keywords: Malware · distributed execution · anti-virus · EDR · injection · code reuse · application hardening· ROP · TxF · WDEG.

1 Introduction

Malicious software is a plague on users and organizations as it can compromise the availability and integrity of computer systems. To shield machines from the overwhelming amount of new threats appearing every year, anti-malware solutions devise dynamic analyses to flag untrusted software as potentially malicious by monitoring its execution traits. In particular, Anti-Virus (AV) and, more recently, Endpoint Detection and Response (EDR) solutions protect home and business computers by combining traditional signature-based mechanisms for binaries with behavioral detection techniques to forestall new threats.

In principle, threat actors may bypass behavioral analysis by diluting the temporal and spatial features of a malicious computation in multiple execution units. This approach requires partitioning a payload into coordinated components so that no one of them causes an AV/EDR system to raise an alert [40].

Some academic literature [40,26,21,8] explores distributed malware execution designs that, using manual or automated methods, craft components that execute as independent processes and coordinate between themselves, possibly through covert channels. Using dedicated processes as units, however, is a conspicuous trait that exposes every process to immediate analysis and, depending on the ignition method, to correlation attempts from security solutions. Even recent designs where each process imitates benign applications in a mimicry fashion [8] need to create no less than 18-20 processes to avoid detection.

Two recent proposals [20,37] make a leap forward by distributing a payload across pre-existing, benign processes through the use of process injection techniques. Such designs bring a strictly harder problem for defenders [20], as they need to correlate actions (e.g., API calls) that are spread out in the event streams for both the victim processes and the entire system. Unfortunately, as we elaborate in more detail later, both designs exhibit conspicuous features and build on primitives that make them an easy prey of state-of-the-art AV/EDR solutions. Also, they both conflict with modern operating system (OS) mitigations that users can enable to harden applications against subversion attempts.

Our approach. In this work we introduce Rope, a new covert design for multi-process malware execution. Rope meets real-world requirements for deploying distributed malware on victims where both behavioral analyses and application hardening mitigations are in place. To this end, Rope builds on commodity OS features (Transactional NTFS) and attack techniques (Return-oriented Programming) for covert communication and for payload encoding and distribution, minimizing the footprint and in turn the conspicuous features of the execution runtime. Rope further raises the bar for defenders as threat actors can replace individual components of the implementation with alternative primitives.

In our tests, Rope eludes the latest version of state-of-the-art AV/EDR products and complies with common opt-in hardening mitigations available on Windows 10. As part of a responsible disclosure process, we reported to Microsoft three flaws (and bypasses) for Windows Defender Exploit Guard and cooperated with one vendor to extend their EDR product so as to detect the implementation solutions that we use. Finally, the paper points out several directions and practical remediations to anticipate and mitigate threats like Rope.

Contributions. In summary, this paper proposes the following contributions:

- an analysis of the challenges that lie along the way to practical multi-process execution of malware;
- a new design, Rope, to meet such challenges using commodity means;
- an evaluation of Rope in the presence of state-of-the-art security solutions and application hardening mitigations, where Rope successfully eludes both.

2 Background

This section details fundamental traits of anti-malware defenses for systems and applications, and depicts the state of the art in distributed malware execution.

2.1 Defenses for Systems and Applications

To detect an incoming untrusted program as malicious, anti-malware solutions use two primary techniques [39]: signature scanning and behavioral analysis.

Signature scanning looks for distinctive patterns in the binary representation of a program, and is challenged both by new threats and by obfuscation and polymorphism applied to known malware strains [35]. Behavioral analysis attempts a controlled execution in emulators [3] and even in the real system by shepherding the use of specific APIs. Unlike signature scanning, it can periodically kick in for long-running programs, e.g., as soon as those exercise operations red-flagged and monitored by the anti-malware solution. In a broader connotation of behavioral analysis, we include also in-memory scanning techniques that dynamically look for patterns in the code and data of processes.

While the inner workings of AV/EDR solutions are undisclosed, prior research [15] reports on behavioral analysis concepts involving feature extraction for sequences of performed API calls, graph representations that capture relations also between their parameters or return values, and variants of these ideas. Correlation is usually limited to single execution units or to their descendants.

Typically, on Windows systems the monitoring of AV/EDR solutions operates through hooks for user-space APIs and minifilters for I/O events, while kernel-level hooks that vendors used to apply on, e.g., the System Service Descriptor Table are no longer allowed since the introduction of PatchGuard [27].

We also note that EDR solutions are nowadays popular among enterprise users as they embody a multifaceted approach, extending the protection of AV engines with improved monitoring capabilities (e.g., remote telemetry) and complementary solutions (e.g., whitelisting, threat detection and response).

Recently available OS mitigations for hardening applications can then serve as a further line of defense. Introduced with Windows 10 version 1709, Windows Defender Exploit Guard (WDEG) blocks several behaviors commonly used in malware attacks [29], offering orthogonal protection alongside AV/EDR solutions. WDEG supersedes and widens the exploit-oriented protection of Microsoft EMET with a number of opt-in hardening mitigations [28], including:

- **ACG** (*Arbitrary Code Guard*), to block the allocation of executable memory as well as permission changes to host dynamically generated code;
- **CIG** (*Code Integrity Guard*), to block the loading of non-signed code;
- **EAF** (*Export Address Filtering*) and **IAF** (*Import Address Filtering*), to block code not from a disk-backed module when accessing the export and import address table of any loaded module (e.g., to locate API addresses);
- **ROP** mitigations (*CallerCheck*, *SimExec*, *StackPivot*), to validate upon an invocation of a sensitive API its call and return sites and the stack pointer value against typical traits of return-oriented programming attacks.

2.2 Distributed Malware

The idea of using multiple processes to deliver a malicious computation dates back at least to 2007 with the formal model of k-ary malicious codes by Filiol [13], and has seen multiple uses in the wild over the years [15]. As we mentioned in Section 1, a rather rich academic literature explores variants of this idea: initially to thwart signature scanning [39], but soon enough behavioral detection became the main target (e.g., [26,21,15]). Multi-process execution can be effective against behavioral analyses as their "dynamic" signatures often involve sequences of events that are not short (or the risk of false positives could be very high [26]): henceforth attackers may spread smaller parts to separate execution units.

Most distributed designs require the creation of new processes, which brings a few practical problems. As several AV/EDR products can look for correlations among processes and their descendants as a form of spatial locality [40], a distributed runtime should try to spawn each execution unit as a sibling process of the others, which is non-trivial to achieve. Also, untrusted executables may run with low integrity levels or be subject to restrictions for, e.g., network access. Recent studies in ransomware design [8] suggest that a high number of processes is required to avoid special-purpose detectors—we picture those as part of an EDR ecosystem—even when attempting mimicry strategies.

Researchers have also explored how to leverage existing processes instead, an approach that brings several benefits at once if successful. For instance, the malicious actions get further spread among the own activities of each victim application, and an execution unit can inherit the access rights of the victim to get around, e.g., application whitelisting or Egress filtering policies [31].

Process injection techniques may offer an avenue to this end. malWASH [20] chops an existing binary into chunks of variable length (e.g., one per basic block) and, from a loader component, injects them into pre-existing processes along with an emulator that orchestrates their execution from a dedicated thread. D-TIME [37] ameliorates malWASH in two respects: it replaces the (conspicuous) remote thread created for emulator dispatching with a mechanism that injects Windows APC calls, and simplifies the communication channel creation.

Unfortunately, both systems would struggle with deployment requirements as we consider the sophistication of ever-improving defenses. malWASH and D-TIME place their emulators and chunks in executable memory regions: a design trait that is fundamentally incompatible, depending on the injection method, with either ACG or CIG-like policies. Their very same complex emulator ($5,500$ hand-written assembly lines in malWASH, which need orthogonal diversification techniques to avoid fingerprinting) is conspicuous, as it uses multiple shared regions to host data segments, heap, and stack for the payload, and others for metadata required to coordinate chunk execution. As we will see later in the paper, these and other practical traits make even such recent designs an easy prey of sophisticated AV/EDR solutions.

3 Challenges for Covert Distributed Malware

This section identifies as challenges a number of requirements and properties, missed in prior works, behind covert designs and implementations of distributed malware abstractions. We believe that, in the lack of a principled approach to cope with real-world deployment requirements, such abstractions would end up being thwarted by narrow-scope remediations from anti-malware solutions, as already happened even for promising concepts like malWASH and D-TIME.

While the list presented in the following may not be exhaustive, we hope it can advance the knowledge in malware design and favor the development of comprehensive mitigations for upcoming threats. We assume a generic design skeleton where an initial component, the *loader*, initiates the distribution of the payload among the leveraged execution units. Each unit executes pieces (*chunks*) of the whole computation with the assistance of a local *bootstrap* component. Due to the inherent limitations of process creation-based designs (Section 2), our main focus for the discussion will be injection-based designs, but the considerations we are about to present are mostly general and apply to both.

[C1] Use a flexible delivery technique. This point is important especially for injection-based designs, as threat actors and researchers regularly come up with new injection techniques [23] to slip through the cracks of evolving AV/EDR solutions and OS mitigations. The covertness of the method used to deliver the chunks to execution units can be almost as important as its ease of replacement, which in turn can also favor the diversification of malware instances.

[C2] Minimize the footprint of the runtime. There are at least two factors that contribute to making a distributed runtime conspicuous: the size of the bootstrap component and the dispatching of the chunks for execution.

The first aspect is a proxy of the complexity of the coordination tasks that the component implements. A small size for it brings a less conspicuous footprint against in-memory detections and fingerprints. Depending on the delivery technique, it may even be accommodated in caves of benign modules. The engineering effort in malWASH to support the execution of already existing payloads is commendable, yet we would argue for a less complex runtime eased by payload writing choices. Compatibility is an important requirement, for instance, for obfuscations meant to protect legitimate software, but in the context of malware we find it fair to assume that a threat actor would be willing to comply with the runtime to ease the payload chopping and coordination processes.

The second aspect involves spreading the actions from the chunks among those of a whole process[1]. Having the loader—or the bootstrap—component attempt the creation of remote threads (malWASH) or entries in the APC queue of a process (D-TIME) are actions that trigger the real-time behavioral monitoring components of AV/EDR solutions. A covert design may instead attempt to make the process itself dispatch the execution of the chunks, ideally choosing among multiple primitives or even diversifying them among the execution units.

[C3] Ensure compliance with hardening mitigations. This point affects the loader and the bootstrap components, the placement of chunks, but also specific actions from distributed payloads. Avoiding the use of RWX regions (or 'W$\oplus$X' permission changes) is essential in the presence of hardening mitigations. It also has benefits against behavioral detection, as dynamic code changes are a

[1] This is relevant also for mimicry attacks from process creation-based approaches.

prerogative of specific classes of applications, such as browsers, that a security solution can whitelist. Similarly, covert strategies used in either component or in the payload to locate APIs covertly need retrofitting work so as not to conflict with mitigations like EAF and IAF meant to block such attempts.

[C4] Keep code and data hidden as much as possible. Along with [C1,2], this point is instrumental to minimize both the indicators of compromise on a machine and to which extent the distributed execution is exposed to in-memory inspection. Ideally, the runtime may expose the code and data for a payload portion only for the fraction of time needed for its execution. As noted in [20], leaving the bootstrap component visible in memory may possibly make defenders move towards detecting the distributed mechanism instead of the target payload, but has the advantage of hiding the true functionality of the latter. Special care is advised also for covert communication channels, so as to hinder the inspection of their contents through, e.g., dynamic signatures from behavioral analysis.

[C5] Limit the footprint of any unavoidable suspicious action. While suspicious behaviors from a payload may be effectively spread out to multiple execution units, there are actions carried in the loader or the bootstrap component that may alert a behavioral detection. Their design may thus seek to reduce the extent of information that an AV/EDR solution receives for their actions by, for instance, circumventing API hooks in place. While such anti-analysis tricks may not suffice to deceive a behavioral analysis of a standalone full payload, as AV/EDR solutions also access other sources (Section 2.1), they can bring instead marginal gains to the overall stealthiness of a distributed concept.

Discussion. The five challenges depicted above are meant to capture the operation of present defenses, and also possible straightforward combinations or extensions of current detection capabilities and fingerprints. Hence, they may be seen as forward-looking targets. Nonetheless, as we detail in Section 6, present AV/EDR solutions already block recently proposed distributed concepts.

Alongside covertness aspects, other issues exist for supporting a distributed execution paradigm. Thankfully, prior literature tackles such aspects well. These include, for instance, feature distribution based on process characteristics and restrictions [31], split communication channels so as to expose only pairwise links between units [26], object marshaling and descriptor duplication [26,20], resilience to termination of single units [20], and drop-in replacements for APIs that provide or work on process-specific information [20,37]. Our Rope concept is orthogonal to such research and can fully benefit from it in an implementation.

4 Rope

In this paper we propose a novel design, Rope, to advance the state of the art in distributed malware execution and overcome the challenges of Section 3. To this end, we pursue a radically different approach compared to prior works.

Rope uses a standard attack technique, return-oriented programming [41] (ROP), and a commodity Windows feature, Transactional NTFS [1,42] (TxF), to

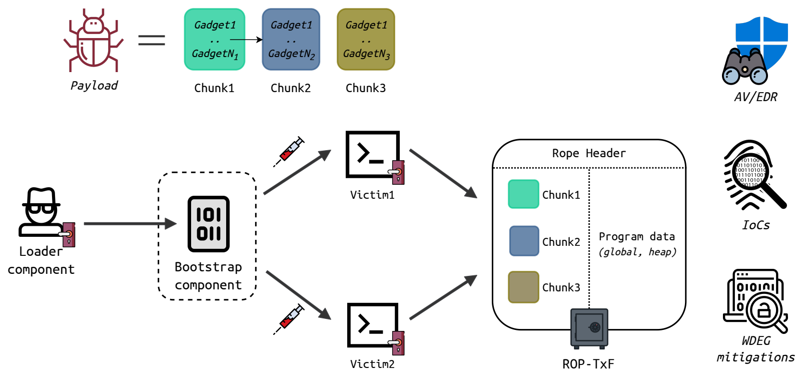

Fig. 1: Architecture of Rope: chunks, runtime components, defenses in place.

encode the required computations and distribute them through a covert channel that also hosts program data. Rope sidesteps the issue of having to allocate or modify executable memory, and uses ROP as an effective and versatile tool to bypass other hardening mitigations of Windows and to raise the bar against in-memory inspections. Rope comes with a small runtime footprint and is amenable to different implementation choices and delivery techniques, as well as to known obfuscation and diversification techniques for both its ROP and non-ROP parts.

Before presenting the general traits of the design, we define which are the defenses that we assume to be in place on a machine where Rope is deployed.

Attack target. We target a machine equipped with an AV/EDR solution with behavioral detection. No previous threat compromised the machine, as the detection capabilities of the AV/EDR could otherwise be affected. The targets for injection by Rope are already running processes that may have opted in for hardening mitigations conceptually akin to[2] the WDEG ones listed in Section 2.1.

The exploitation of a human or system vulnerability necessary to ignite the loader component, and the evasion of possible fine-grained analyses (e.g., malware sandboxes) along the way to a victim, are orthogonal problems. Rope is compatible with existing means for achieving both ends; the goal of Rope and prior works in the area [20] is that the execution stays undetected on the victim.

4.1 Architecture

Figure 1 presents the architecture of Rope. As in the generic skeleton for distributed malware depicted in Section 3, we use a loader component to infect a set of (pre-existing) victim execution units. We then deliver the bootstrap component to each victim using an injection primitive. Rope encodes this component as a ROP chain. However, if the injection primitive is capable of eluding CIG (we discuss one in Section 5), also a short shellcode is a possibility. The malicious

[2] Necessary to overcome present implementation gaps in WDEG, e.g., with concomitant use of ACG and ROP mitigations [28], or with ACG and remote allocations [19].

payload comes as a sequence of chunks expressed as ROP chains. Along with runtime metadata, we lay them out in a transacted NTFS file: the ROP-TxF.

Transactional NTFS is a Windows feature to atomically manipulate files and directories, where a series of operations (including content changes) can form a transaction to commit or abort as a whole in the surrounding system. Kulkarni and Jagdale observe in [24] how this provides a form of read-committed isolation. Until a transaction is committed, transient changes are visible only to the processes that created or obtained a handle to the transaction (and to the Windows Kernel Transaction Manager that backs the TxF functionality [1]). AV/EDR products can make a minifilter driver enlist for TxF commit events (e.g., to detect a TxF-based strategy to infect files on disk [24]); however, any content transiently stored in a TxF file will stay hidden to them, and Rope does not need commit operations to support its distributed execution.

The loader component of Rope creates the ROP-TxF on a random file, alters its contents, and duplicates a handle for it for every victim process. Each instance of the bootstrap component uses its own duplicate handle to access and modify the ROP-TxF contents, dispatching the execution of the chunks in it in a coordinated manner. The Rope paradigm is general enough to allow for (at least) the following execution modes, which may also be mixed in the same payload:

- one follows a *feature-agnostic* scenario where the Rope runtime, similarly as in malWASH and D-TIME, aids in explicit coordination of the chunks without making distinctions about the characteristics of their actions;
- the other follows a *feature-aware* scenario: the Rope runtime offloads sequences of chunks to victims based on their capabilities and/or with a separation in functionally distinct units, similarly to malware seen in the wild.

In the following we detail the different components and how Rope meets the [C1-5] challenges on the way to covert multi-process malware execution.

4.2 Loader Component

The loader component initiates the distributed execution by first looking for preexisting processes to target as victims. Compared to standard injection practices from malware, Rope does not add any special requirement for process selection. In the feature-aware scenario, the malware writer can specify preferred targets (for instance, browsers are suitable for chunks that need to establish network connections). The loader component has then to accomplish three main tasks:

1. creates the ROP-TxF and duplicates the TxF handles for the victims;
2. injects the bootstrap component in each victim;
3. tampers with each victim to dispatch the bootstrap component.

The first task is straightforward. For the second task, we must avoid the use of any executable memory. As we will detail when presenting a possible implementation in Section 5, we inject a short ROP chain to have the victim process itself load the bootstrap component. This chain also accomplishes the third task, choosing among different dispatching strategies. In more detail, we may make the victim create an own thread or schedule an APC (neither of which is suspicious, unlike the remote variants used in malWASH and D-TIME), or we may tamper with its IAT to realize an API call target hijacking.

The implementation can use off-the-shelf techniques to place the chain in the address space of a victim. A reader may wonder if such an action should be blocked by a behavioral detection, or if it may undermine the stealthiness of the approach. Denying the use of `WriteProcessMemory`, `NtWriteVirtualMemory`, or other commodity OS features (e.g., Atom tables) that malware abuses as write-what-where primitives may break the functioning of legitimate applications. Such an action should definitely be part of behavioral signatures but, most likely, together with other conspicuous criteria to be met. Also, techniques like Ghost Writing can write to a process without even having to open it [23]. Nonetheless, Rope may also resort to evasion techniques for API hooks to reduce the footprint of specific actions: we defer their discussion to Section 5.

Similarly, no Rope-specific provisions are necessary to make the victim execute the initial short ROP chain: for instance, we may hijack a thread, overwrite a return address in a stack frame, or use other standard techniques.

4.3 Chunk Crafting and ROP-TxF Layout

Rope is the first design to use a covert communication channel for hosting both instructions and program state: as even chunks take the form of data, the execution runtime can host both using a single RW region. We use NTFS transactions to keep the contents of ROP-TxF hidden as much as possible, but as we will discuss in Section 4.5 also other standard means can provide hosting in its place.

As the top part of Figure 1 shows, a payload that Rope spreads out takes the form of a series of ROP chains of user-defined length, with each chain embodying a chunk. In the implementation, we use chunks that encompass a single API call, as in the eyes of behavioral detection API calls are the unit of interaction with the OS. Trade-offs with larger chunks should also be possible [20,37].

Careful choices in payload designing help in keeping the execution runtime small and with fewer conspicuous features. For starters, we want to avoid the creation of multiple memory regions (e.g., stack, heap, data, rodata), each with different permissions and relocations as for the typical segments of an executable. The design point that we follow is radically simple: we model global, heap, and stack memory as part of a flat space in the ROP-TxF. We promote each stack variable[3] to global storage, and we encapsulate all global variables and objects (e.g., strings) as fields of a single data structure. Every access to a program variable undergoes a level of indirection in a base+offset computation, and upon loading the ROP-TxF we control the value that the runtime exposes for base.

To encode ROP chains, we can pick gadgets from a code module that the victim processes share already before or as a consequence of the actions of the

[3] Only recursive functions would require semantic changes to their code: e.g., the attacker may use a stack data structure to host and reference each stack frame.

loader or the bootstrap component. Another important issue in chunk crafting is the sharing of OS resources among execution units. Well-studied in [20], this issue involves Windows `HANDLE`, registry, and socket objects as they are unique per process (i.e., other units cannot access them). Since we already wrap program variables, we can identify the involved handles and then duplicate them via standard OS features, exposing to each unit its own copies.

We can now detail the layout of the ROP-TxF, which hosts three adjacent areas: *header*, *chunks*, and *program memory*. The header aids the runtime in bookkeeping tasks. It holds the index of the last executing unit and of the current chunk, the addresses of the Windows APIs that the payload may use, and a per-unit structure hosting duplicated handles (for TxF and program resources), scratch locations for ROP gadgets, and other runtime metadata (e.g., for the feature-aware scenario). Chunks are indexed and laid out consecutively, then program memory follows as it can expand by appending to the ROP-TxF file.

4.4 Bootstrap Component

The bootstrap component serves two main purposes: scheduling the chunks in coordination with the other units and loading the ROP-TxF to memory.

For execution coordination, Rope can benefit from well-studied techniques to implement locks in increasingly covert ways [20,37,6]. Once a victim execution unit acquires the lock, it updates the ROP-TxF header and advances chunk execution. Upon execution termination, the bootstrap component may actively wait for new chunks to process or go dormant for some time. In the latter scenario, the bootstrap component arranges for its rescheduling if the loader used a method with a limited lifetime (e.g., an APC call—Section 4.2) to dispatch it.

As for the ROP-TxF loading, we identify two alternative strategies:

a) Use the ROP-TxF as a memory-mapped file and access its contents directly. Any change will be readily visible to the other units. The mapping may be active persistently or recreated whenever about to execute a chunk.
b) Read the strictly needed portions from the ROP-TxF file and copy them to process-local memory (e.g., heap), then write them back to propagate any updates (e.g., program state values) to the other units.

The two strategies come with a different trade-off between stealthiness (i.e., visibility of the full ROP-TxF contents to in-memory scanning techniques) and bookkeeping work (i.e., explicit synchronization of contents). Present AV/EDR solutions do not appear to be equipped with ROP-aware analyses that may benefit from any available memory contents. Hence, we believe that for now, a malware writer would choose one of the two based on factors beyond the goals of Rope, such as trying to hinder forensic analysis during incident response.

Irrespective of such choice, the functionality that the component will enact is the same: it changes the stack pointer to point to the beginning of the first chunk to execute, dispatches one or more chunks, and eventually reacquires control.

Finally, we give the bootstrap component a further and more practical task: solving in a covert way the APIs used in the payload, filling currently unassigned address fields in the ROP-TxF header. While handling API resolution in the loader would be simpler (e.g., by importing such APIs statically or by solving them dynamically), it would give away hints on the actions (e.g., crypto, network) that the payload may perform. Also, it may get in the way with more complex payloads where, e.g., a sample receives from the network a functional update that exercises APIs not initially considered. Instead, the processes that Rope abuses typically use the enclosing DLLs already for their activities. Therefore, the bootstrap component can scan their memory covertly to look up APIs.

4.5 Discussion

Rope brings new advances to the malware design literature by meeting the five open challenges we identified along the way to covertness (Section 3) with original solutions. These include, among others, the flattening of code and program data as relocatable elements of a single region, the covert dispatching of the bootstrap component as part of the own activities of the process, and forfeiting the introduction of executable regions and other forms of dynamic code generation.

Along with careful design choices for the components of the architecture, code reuse techniques turn out to be decisive to achieve all these ends and properties. The means to deliver chunks to execution units sees no Rope-specific constraints [C1], and we do not even inject code but data (Section 4.2). The bootstrap component, thanks to payload crafting choices such as indirection for data accesses and clear fields for handles (Section 4.3), is conceptually simple and in turn compact [C2]. As Section 5 will detail for [C3], ROP offers a means to bypass hardening mitigations like EAF and IAF and to make Rope comply with others by design. The loading strategies for the ROP-TxF contents and the use of TxF can go a long way for [C4]. Finally, as we observed in our experiments, only the actions of the loader may need special countermeasures to limit their footprint [C5]. Indeed, unlike for the bootstrap component and the chunks, we cannot disguise its actions as if they were from pre-existing processes.

Another design advantage of Rope is that the technical means that we use are virtually interchangeable. For instance, we may replace or combine ROP with other code reuse techniques, and use different methods in each victim to dispatch the bootstrap component. We may possibly replace even the ROP-TxF using a standard file or a shared memory, albeit with a loss of covertness for its contents against minifilters from AV/EDR solutions or in-memory scans. As we will argue in Section 7, defenders should equip with comprehensive detections for distributed malicious executions, as mitigating only individual components may turn out insufficient.

On a different note, a few research works have explored uses of ROP in malware design, e.g., for split-personality [44] and polymorphism [34] schemes. Those works make a surgical use of ROP to target weaknesses in static code analysis schemes: both the adversary (e.g., AV signatures in [34]) and the extent of the ROP encoding (i.e., limited to few parts) are different than in our work.

To the best of our knowledge, Rope is the first work successfully targeting Windows systems hardened with AV/EDR solutions and the opt-in mitigations of WDEG. The design behind Rope is general: we believe that with implementation adaptations one may explore it also on other platforms, such as Linux [16] and Android [22], for which injection techniques recently started to emerge.

As anticipated at the end of Section 3, when presenting the design of Rope we did not touch on aspects such as split communication channels or resilience to termination, as for those problems Rope can adopt existing solutions.

5 Implementation

This section illustrates the choices that we followed in a prototype implementation of Rope, and reports on the bypass techniques for the WDEG mitigations that we created and responsibly disclosed to Microsoft in February 2021. A companion technical report [10] covers details of the implementations of both Rope and the bypass techniques that we omitted in the following for brevity.

Injection and Gadget Source. As described in Section 4.2, the loader component needs to inject in each victim process a short ROP chain to make it load the bootstrap component of Rope. To this end, we resort to a standard technique in malware for hijacking one of the threads of the victim. We retrieve the CPU context, alter the stack pointer to make room for the chain, write the chain to the victim's stack along with the saved CPU context, and update the instruction pointer so as to execute the first gadget. Upon thread resumption, the chain loads the bootstrap component, dispatches it for execution, and eventually restores the saved CPU context to resume the victim's activities.

In Section 4.3 we also discussed the possibility of using as a gadget source a DLL module that all victims load as a consequence of the actions of either the loader or of the bootstrap component. In the implementation, we foresaw an opportunity for a shortcut to ease the loading of both the bootstrap component and such DLL module: when TxF-ed files are involved, the CIG mitigation of Windows struggles with operations that are not self-contained. In particular, while WDEG causes a checksum error for `NtCreateSection` if a CIG-enabled process tries to map a section object from a TxF-ed file, it currently does not stop it from mapping a view of a section object created in another process.

Hence, we make the Rope loader create another transacted file on a signed Microsoft DLL, add to it any required gadgets using code caves, and embed also the bootstrap component (as a ROP chain or even as a shellcode placed in RX caves) in it[4]. Then we create a section object and duplicate the handle to it.

The initial ROP chain that we inject uses 6 one-instruction gadgets from `ntdll.dll`. To make the victim load the transacted DLL it uses the system call `NtMapViewOfSection`, as the ROP mitigations of WDEG do not shepherd it. Then it executes a helper sequence of the bootstrap component that creates a thread for it to run, and readily returns to the chain.

[4] A more conservative and covert implementation may target an already-loaded module (e.g., `kernel32.dll`) and encode the bootstrap component and the chunks with, e.g., microgadgets [17] that are abundant [32]. However, this was not necessary for validating the stealthiness of our approach on the currently available tested defenses.

The bootstrap component uses the ROP-TxF as a memory-mapped file and a simple mutex to coordinate the execution. As the transacted DLL can see different base addresses due to the non-standard loading that we use, before executing a chunk the component applies relocations to gadget addresses and saves the current base address for the DLL in the ROP-TxF header. In this way, the next execution unit can perform its own relocations with the same technique.

API Calls. The interaction with APIs from the OS brings three problems: locating them covertly in the presence of EAF/IAF mitigations, eluding the ROP mitigations of WDEG for sensitive APIs, and evading AV/EDR hooks.

The EAF and IAF mitigations (Section 2.1) monitor accesses to the export and import address tables of loaded code modules: those are recurring traits in covert API resolution methods. To avoid suspicious imports in the loader or uses of noisy OS means (i.e., `GetProcAddress`) for API resolution, we designed and responsibly disclosed to Microsoft two bypass techniques for EAF and IAF, respectively, that the bootstrap component can alternatively use. In both techniques, we disguise accesses to the export/import address table of the victim program by using a ROP gadget that can read from an arbitrary address, searching it within any loaded DLL (e.g., `kernel32.dll`). The current implementations of EAF and IAF whitelist such an access, erroneously assuming it originated in a legitimate module. Further details can be found in [10].

Our implementation complies with the WDEG ROP mitigations for sensitive APIs (Section 2.1) that a chunk or the bootstrap component may invoke. In particular, WDEG shepherds invocations of several APIs that are typically used for manipulating memory, processes, and code modules [28]. To get around these mitigations, for *StackPivot* we switch the stack pointer so as to have any API calls (i.e., also non-sensitive ones) take place on the native program stack, while for *CallerCheck* and *SimExec* we use suitable gadgets described in prior works [33,5].

Finally, eluding API hooks may be useful for setup activities that may arouse suspicion in a behavioral engine [C5]. In our tests, only the loader seemed to benefit from hook evasion: indeed, due to its standalone and intrusive nature, the loader can be the weak link in distributed malware concepts. Our implementation may call standard high-level APIs, use their low-level `Nt` counterparts from `ntdll.dll`, or attempt hook evasion. On 64-bit victims, "direct" system calls [11] can elude user-space hooks for `Nt` functions by solving the ordinal for each system call and invoking it with an own ASM stub. However, for 32-bit victims evasion is harder in Windows 10, as system calls (like the whole kernel) are 64-bit code and take place in the WOW64 compatibility layer. As evasion strategy we opted to use the `wow64ext` helper library[5] to make a transition (colloquially known as "Heaven's gate") into 64-bit code and call the 64-bit `Nt` functions directly, as AV/EDR products may neglect them. As future work, we would like to stress such security solutions further by issuing direct system calls after Heaven's gate.

Payload Encoding. Manually encoding a ROP payload can be a lengthy task [2]. The current implementation assumes that the attacker encodes the payload as one or more C functions, delimits the chunks manually (e.g., with

[5] https://github.com/rwfpl/rewolf-wow64ext.

an inlined ASM `nop`), and compiles it to object code using a standard compiler (we used Visual Studio 2019). With careful compilation settings [12], such as omitting the generation of stack canaries, the output resembles a position-independent code for which we can partially automate the ROP translation.

In more detail, we look up or introduce suitable gadgets in the target DLL for one instruction at a time, and wrap constructs like conditional transfers and API calls with templates. Promoting stack variables to global storage in the design sidesteps difficulties that would emerge here and that are known in the ROP practice, as in general translating stack manipulations may require non-trivial program analyses or the use of a parallel stack [4]. While implementing a fully automated translation goes beyond the scope of proving the effectiveness of Rope in avoiding detection, we believe it is a realistic goal also in light of the recent advances in ROP-based program obfuscation [4,32]. Also, capable attackers have already written and released fully-ROP malware in the past [14,9].

6 Evaluation

This section details the findings of our experimental validation. First, we present the methodology for testing AV/EDR products and the WDEG mitigations, and illustrate the PoC Rope samples that we exercised. We then report on our findings and on interactions that we had with a vendor of a popular EDR solution.

Design of Experiments. As the inner workings of commercial AV/EDR solutions are undisclosed and fingerprinting them through manual reverse engineering is difficult [3], we follow a black-box approach to test the success of the covert distribution. We craft several standalone payloads involving different tasks typical of malware (e.g., download and execute) and verify that each security solution would detect them as malicious via behavioral analysis. We then encode equivalent sequences in Rope and test if they avoid detection as expected. For presenting the results, we chose one minimal representative sample for either Rope execution mode (i.e., feature-agnostic and feature-aware, Section 4.1).

As for WDEG mitigations, we test different combinations to rule out possible interferences among them: we found one between ACG and IAF and reported it. Also, for each employed bypass technique we verified that a violation was indeed detected for an alternate payload that did not make use of it. To rule out false positives from implementation gaps, we test the mitigations in audit mode and review the Windows Event Logs for any report originating from the own activities of the application. These may happen, e.g., when an application loads a legitimate module that was not signed by Microsoft or the Windows Store, or when a browser uses dynamic code generation without liaising with ACG/CIG.

Subjects. For testing, we use three versions of Windows 10 (2004, 20H2, and an Insider build from January 2021) running in a VMware Fusion appliance. All versions come with the following system-wide mitigations enabled: DEP, ASLR, Mandatory ASLR, Bottom-up ASLR, High-entropy ASLR, SEHOP, Validate Heap Integrity, and CFG [28]. We pick our victims from 9 popular applications

	Security solution	Version
AV	Avast	2.1.27.0
	Bitdefender Total Security	25.0.10.52
	Kaspersky Total Security	21.2.16.590
	Intezer Endpoint Scanner	1.0.1.8
	Malwarebytes	4.1.1.167
	Windows Defender	January 2021
EDR	Comodo Client Security	12.5.0.8351
	Microsoft Defender ATP	January 2021
	Sophos Intercept X	2.0.18
	Webroot SecureAnywhere	9.0.29.62

Application	Version
Adobe Reader DC	19.010.20098
Chrome	86.0.4240.198
Discord	0.0.309
Dropbox	112.4.321
Firefox	83.0
Opera	73.0.3856.257
Steam	2.10.91.91
Skype	8.66.0.77
Telegram	2.4.7

Table 1: Tested AV/EDR solutions and victim applications.

of varying complexity (Table 1) in their 32-bit releases to allow for a direct comparison with prior solutions, and we make them opt-in for (combinations of) the mitigations that Rope targets (ACG, CIG, EAF, IAF, and ROP—Section 2.1).

As state-of-the-art security solutions, we evaluate Rope on 6 AV and 4 EDR products (Table 1) for which we could obtain a trial version. We run each application and the Rope loader with medium integrity level. We use distinct VM instances for each product, and restore their state after each test.

For the feature-agnostic mode, we use a payload that tampers with the Windows registry by introducing a key entry to achieve persistence for the loader or, alternatively, to invoke a privileged system utility like `bcdedit`. The victim processes (two already sufficed) can race in any order to execute every chunk. For the feature-aware mode, we create a download-and-execute scenario where one network-active process executes chunks that download a PowerShell script from a remote C2 server and drop it to the `%TEMP%` folder, while a normal process executes chunks that read the file path from the ROP-TxF and run the script.

Results. None of the security solutions detected Rope, and no alerts were raised for WDEG as expected. In more detail, we found that the injection primitive that we use in the loader went unnoticed by 7 of the 10 products even when using high-level APIs. One product instead aggressively flags any attempt to open a process and write to its memory: calling the 64-bit `Nt` functions with Heaven's gate suffices to elude it. Finally, two products cause errors (`5: Access Denied`) when trying to open a process. By further investigation, we observed that both `OpenProcess` and `DuplicateHandle` see flawed outputs and errors. This behavior is coherent with prior research on inconsistencies in AV emulators [3]; also, it seems to suggest that these security solutions initially run an untrusted binary as with low integrity and/or in a sandbox-like fashion, shepherding its interactions with the OS in the early execution stages. Unfortunately, direct system calls and other evasions may be effective in defeating even this approach.

On a different note, during our experiments we had several fruitful interactions with one of the EDR vendors, who graciously granted us a trial for testing their product with new concepts and eventually extended it to cope with our implementation tactics. The information that they shared with us on their remote telemetry and detections confirmed the truthfulness of our black-box findings.

Finally, to compare with prior research in the field, we distributed the standalone version of the two test payloads using D-TIME: only 3 of the 10 AV/EDR products did not block it. The result is not surprising, as the novelty of remote APC injection wore off shortly after the paper was published. Also, we note that Windows 10 already in its 1809 version added sensors for more complex forms of APC injection (e.g., Defender ATP uses them to detect DoublePulsar-like APC attacks [30]). We do not test malWASH for covertness: while it provides a robust distributed execution runtime, it uses a very conspicuous delivery primitive that D-TIME already improves. In general, revamping either system with more cutting-edge injections would only bring short-lived benefits: vendors will add detections for new injections, while their emulators and chunks will remain an issue in the eyes of AV/EDR analyses and hardening mitigations.

7 Countermeasures and Wrap-up

Our experiments suggest that Rope hits a blind spot in the characteristics of AV/EDR solutions, even in the presence of application hardening mitigations meant to reduce the attack surface for malware. A fundamental question is how to counter this and similar threats in anti-malware defenses running on machines.

Caveat: *As the details of AV/EDR solutions are undisclosed, nuances of the ideas that we outline next may be already present in them in ineffective or incomplete forms, whereas none aided the systems that we tested with* Rope*. Also, we do not claim to cover in the following all possible methods to detect* Rope*.*

From a methodological perspective, we foresee three complementary avenues. The first involves enriching process correlation heuristics in behavioral detection by monitoring primitives that are instrumental to distributed computations. As making a real-time detection scale to multiple units for correlation is computationally expensive [26], heuristics may "draw its fire" to candidate groups of seemingly unrelated processes for close monitoring as a whole. These heuristics should conjecture links by tracking not only process injection primitives (which are continuously evolving and thus hard to cover exhaustively) but also—and possibly with a major emphasis—those OS features that distributed execution designs would need for sharing objects (e.g., handle duplication) and hosting data contents (e.g., TxF, shared memories, memory-mapped files).

The second involves extending the dynamic signatures and in-memory analyses of security solutions with techniques for detecting [38] and analyzing [14,9] payloads encoded with ROP or other weird machines. For instance, a solution may look for repeated gadget sequences used to invoke API calls or for the arrangement of typical constants as API call arguments. On the other hand, attackers may experiment with gadgets more complex than the ones we use in Rope, for instance by playing with gadget diversification and dynamically dead code [4]. We observe that in this setting ROP-aware analyses may serve a more general purpose, as the recent availability of systems for encoding whole programs in ROP for obfuscation [4,32] may also encourage threat actors to experiment more with ROP in standalone malware too (e.g., for polymorphism [34]).

The third involves stronger means to intercept the execution footprint of any kind of malware. The evasion of user-mode API hooks can be a customary practice for the stealthiest malware strains. In addition to minifilters, the Windows kernel provides notifications for events like process creation and termination[6], which AV/EDRs and other systems (e.g., anti-cheat engines) use also to intercept subversion attempts. However, Ciholas et al. show in [7] that a user-mode process can outrun them. A more resilient approach to monitoring application-level events may be using hardware-assisted virtualization features, such as multiple EPT views [25,18] to insert hooks in the physical pages for OS and processes [11].

Albeit the design of Rope leaves much leeway to implementation diversification, stopgap measures may in the meantime help for immediate remediation. As mentioned early on, we reported several flaws in WDEG to Microsoft, suggesting practical countermeasures for each. Adding detection logic to AV/EDR engines based on the implementation solutions detailed in this paper may help as well.

Security solutions may also explore fine-grained detections of ROP execution via hardware assistance. For instance, the HitmanPro.Alert proactive detection system can use the Last Branch Record feature of Intel processors to heuristically detect ROP (as much academic literature explored in the past [36]), albeit false positives may be frequent on, e.g., browsers. Windows 10 with its 2004 version landed instead the first complete implementation of hardware-enforced stack protection by using the Intel Control-Flow Enforcement Technology (CET). We note, however, that the first processor supporting CET became available at the end of 2020, and that CET still leaves room for attacks [43].

On a different note, Rope poses interesting challenges also for reverse engineering and fine-grained analyses. For instance, the combination of ROP with NTFS transactions may be interesting for fileless malware concepts, and forensic analyses may need tools to carve out TxF contents that are not directly mapped in memory. Offline systems like Panorama [45] that perform expensive system-wide information flow analyses may instead be useful to track computations spread among execution units, so as to shed light in a black-box fashion on the internal working of Rope samples or other distributed malware instances.

We hope that the ideas presented in this paper may contribute to raise awareness on the effectiveness and practicality of covert distributed malware attacks, to foster research on defensive countermeasures, and to improve the implementation of current AV/EDR solutions. The technical details omitted from the paper for brevity can be found in our companion technical report [10]. The Rope samples from our tests will be made available to AV/EDR vendors upon request.

References

1. Allred, C.: Understanding Windows file system transactions. In: Storage Developer Conference 2009. SNIA (2009), https://www.snia.org/sites/default/orig/sdc_archives/2009_presentations/tuesday/ChristianAllred_UnderstandingWindowsFileSystemTransactions.pdf

[6] Also the ETW (Event Tracing for Windows) system offers useful tracing capabilities.

2. Angelini, M., Blasilli, G., Borrello, P., Coppa, E., D'Elia, D.C., Ferracci, S., Lenti, S., Santucci, G.: ROPMate: Visually Assisting the Creation of ROP-based Exploits. In: Proc. of the 15th IEEE Symposium on Visualization for Cyber Security. VizSec '18 (2018). https://doi.org/10.1109/VIZSEC.2018.8709204
3. Blackthorne, J., Bulazel, A., Fasano, A., Biernat, P., Yener, B.: AVLeak: Fingerprinting antivirus emulators through black-box testing. In: 10th USENIX Workshop on Offensive Technologies. WOOT '16, USENIX Association (2016)
4. Borrello, P., Coppa, E., D'Elia, D.C.: Hiding in the particles: When return-oriented programming meets program obfuscation. In: Proc. of the 51st Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks. pp. 555–568. DSN '21, IEEE (2021). https://doi.org/10.1109/DSN48987.2021.00064
5. Borrello, P., Coppa, E., D'Elia, D.C., Demetrescu, C.: The ROP needle: Hiding trigger-based injection vectors via code reuse. In: Proc. of the 34th ACM/SIGAPP Symposium on Applied Computing. pp. 1962–1970. SAC '19, ACM (2019). https://doi.org/10.1145/3297280.3297472
6. Botacin, M., De Geus, P., Gregio, A.: "VANILLA" malware: vanishing antiviruses by interleaving layers and layers of attacks. Journal of Computer Virology and Hacking Techniques **15**, 233–247 (2019). https://doi.org/10.1007/s11416-019-00333-y
7. Ciholas, P., Such, J.M., Marnerides, A.K., Green, B., Zhang, J., Roedig, U.: Fast and furious: Outrunning windows kernel notification routines from user-mode. In: Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 67–88. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-52683-2_4
8. De Gaspari, F., Hitaj, D., Pagnotta, G., De Carli, L., Mancini, L.V.: The Naked Sun: Malicious cooperation between benign-looking processes. In: Applied Cryptography and Network Security. pp. 254–274. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-57878-7_13
9. D'Elia, D.C., Coppa, E., Salvati, A., Demetrescu, C.: Static analysis of ROP code. In: Proc. of the 12th European Workshop on Systems Security. EuroSec '19, ACM (2019). https://doi.org/10.1145/3301417.3312494
10. D'Elia, D.C., Invidia, L.: Rope: Bypassing behavioral detection of malware with distributed ROP-driven execution. Black Hat USA (2021), https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Rope-Bypassing-Behavioral-Detection-Of-Malware-With-Distributed-ROP-Driven-Execution-wp.pdf
11. D'Elia, D.C., Nicchi, S., Mariani, M., Marini, M., Palmaro, F.: Designing robust API monitoring solutions. arXiv **abs/2005.00323** (2020)
12. Doniec, A.: From a C project, through assembly, to shellcode (by hasherezade). VX Underground (2020), https://github.com/vxunderground/VXUG-Papers
13. Filiol, E.: Formalisation and implementation aspects of K-ary (malicious) codes. Journal in Computer Virology **3**, 75–86 (2007). https://doi.org/10.1007/s11416-007-0044-2
14. Graziano, M., Balzarotti, D., Zidouemba, A.: ROPMEMU: A framework for the analysis of complex code-reuse attacks. In: Proc. of 11th Asia Conference on Computer and Communications Security. pp. 47–58. ASIACCS '16, ACM (2016). https://doi.org/10.1145/2897845.2897894
15. Hăjmăşan, G., Mondoc, A., Portase, R., CreŢ, O.: Evasive malware detection using groups of processes. In: ICT Systems Security and Privacy Protection. pp. 32–45. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-58469-0_3

16. Hendrick, A.: Fileless malware and process injection in Linux. Hack.lu (2019), http://archive.hack.lu/2019/Fileless-Malware-Infection-and-Linux-Process-Injection-in-Linux-OS.pdf
17. Homescu, A., Stewart, M., Larsen, P., Brunthaler, S., Franz, M.: Microgadgets: Size does matter in Turing-complete return-oriented programming. In: 6th USENIX Workshop on Offensive Technologies. WOOT '12, USENIX Association (2012)
18. Hong, J., Ding, X.: A novel dynamic analysis infrastructure to instrument untrusted execution flow across user-kernel spaces. In: Proc. of the 2021 IEEE Symposium on Security and Privacy. pp. 402–418. SP '21, IEEE Computer Society (2021). https://doi.org/10.1109/SP40001.2021.00024
19. ired.team: ProcessDynamicCodePolicy: Arbitrary Code Guard (ACG). Red Teaming Experiments GitBook (2020), https://www.ired.team/offensive-security/defense-evasion/acg-arbitrary-code-guard-processdynamiccodepolicy
20. Ispoglou, K.K., Payer, M.: malWASH: Washing malware to evade dynamic analysis. In: 10th USENIX Workshop on Offensive Technologies. WOOT '16, USENIX Association (2016)
21. Ji, Y., He, Y., Zhu, D., Li, Q., Guo, D.: A mulitiprocess mechanism of evading behavior-based bot detection approaches. In: Information Security Practice and Experience. pp. 75–89. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-06320-1_7
22. Kaspersky: Dvmap: the first Android malware with code injection. SecureList (2017), https://securelist.com/dvmap-the-first-android-malware-with-code-injection/78648/)
23. Klein, A., Kotler, I.: Process injection techniques - gotta catch them all (Windows process injection in 2019). Black Hat USA (2019), https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf
24. Kulkarni, A.P., Jagdale, P.D.: Adapting to TxF. VirusBulletin (Jan 2010), https://www.virusbulletin.com/virusbulletin/2010/05/adapting-txf
25. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A.: Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In: Proc. of the 30th Annual Computer Security Applications Conf. (ACSAC '14). pp. 386–395. ACM (2014). https://doi.org/10.1145/2664243.2664252
26. Ma, W., Duan, P., Liu, S., Gu, G., Liu, J.C.: Shadow attacks: automatically evading system-call-behavior based malware detection. Journal in Computer Virology **8**(1), 1–13 (2012). https://doi.org/10.1007/s11416-011-0157-5
27. MDSec: Bypassing user-mode hooks and direct invocation of system calls for red teams (2020), https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/
28. Microsoft: Exploit protection reference, https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/exploit-protection-reference?view=o365-worldwide
29. Microsoft: Windows Defender Exploit Guard: Reduce the attack surface against next-generation malware (2017), https://www.microsoft.com/security/blog/2017/10/23/windows-defender-exploit-guard-reduce-the-attack-surface-against-next-generation-malware/
30. Microsoft Defender Security Research Team: From alert to driver vulnerability: Microsoft Defender ATP investigation unearths privilege escalation flaw, https://www.microsoft.com/security/blog/2019/03/25/from-alert-to-driver-vulnerability-microsoft-defender-atp-investigation-unearths-privilege-escalation-flaw/

31. Min, B., Varadharajan, V.: Design and analysis of a new feature-distributed malware. In: 2014 IEEE 13th Int. Conference on Trust, Security and Privacy in Computing and Communications. pp. 457–464 (2014). https://doi.org/10.1109/TrustCom.2014.58
32. Nakanishi, F., De Pasquale, G., Ferla, D., Cavallaro, L.: Intertwining ROP gadgets and opaque predicates for robust obfuscation. arXiv **abs/2012.09163** (2020)
33. Nemeth, Z.L.: Modern binary attacks and defences in the Windows environment – fighting against Microsoft EMET in seven rounds. In: 2015 IEEE 13th Int. Symposium on Intelligent Systems and Informatics. pp. 275–280. SYSY '15 (2015). https://doi.org/10.1109/SISY.2015.7325394
34. Ntantogian, C., Poulios, G., Karopoulos, G., Xenakis, C.: Transforming malicious code to rop gadgets for antivirus evasion. IET Inf. Security **13**(6), 570–578 (2019). https://doi.org/10.1049/iet-ifs.2018.5386
35. Or-Meir, O., Nissim, N., Elovici, Y., Rokach, L.: Dynamic malware analysis in the modern era - A state of the art survey. ACM Comput. Surv. **52**(5) (Sep 2019). https://doi.org/10.1145/3329786
36. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: 22nd USENIX Security Symposium. pp. 447–462. USENIX Security '13, USENIX Association (2013)
37. Pavithran, J., Patnaik, M., Rebeiro, C.: D-TIME: Distributed threadless independent malware execution for runtime obfuscation. In: 13th USENIX Workshop on Offensive Technologies. WOOT '19, USENIX Association (2019)
38. Polychronakis, M., Keromytis, A.D.: ROP payload detection using speculative code execution. In: 2011 6th Int. Conference on Malicious and Unwanted Software. pp. 58–65. IEEE Computer Society (2011). https://doi.org/10.1109/MALWARE.2011.6112327
39. Ramilli, M., Bishop, M.: Multi-stage delivery of malware. In: 2010 5th Int. Conference on Malicious and Unwanted Software. pp. 91–97 (2010). https://doi.org/10.1109/MALWARE.2010.5665788
40. Ramilli, M., Bishop, M., Sun, S.: Multiprocess malware. In: 2011 6th Int. Conference on Malicious and Unwanted Software. pp. 8–13 (2011). https://doi.org/10.1109/MALWARE.2011.6112320
41. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur. **15**(1) (2012). https://doi.org/10.1145/2133375.2133377
42. Russinovich, M., Solomon, D.A.: Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition. Microsoft Press (2009), pp. 965-974
43. Sun, B., Liu, J., Xu, C.: How to survive the hardware-assisted control-flow integrity enforcement. Black Hat Asia (2019), https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf
44. Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on iOS: When benign apps become evil. In: 22nd USENIX Security Symposium. pp. 559–572. USENIX Security '13, USENIX Association (2013)
45. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing system-wide information flow for malware detection and analysis. In: Proc. of the 14th ACM Conference on Computer and Communications Security. pp. 116–127. CCS '07, ACM (2007). https://doi.org/10.1145/1315245.1315261