# Measuring and Defeating
# Anti-Instrumentation-Equipped Malware

Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo
Fontana, Fabio Gritti, and Stefano Zanero

DEIB, Politecnico di Milano, Italy
{mario.polino,andrea.continella,stefano.zanero}@polimi.it
{sebastiano.mariani,stefano.dalessio,lorenzo2.fontana,fabio1.gritti}@mail.polimi.it

**Abstract** Malware authors constantly develop new techniques in order
to evade analysis systems. Previous works addressed attempts to evade
analysis by means of anti-sandboxing and anti-virtualization techniques,
for example proposing to run samples on bare-metal. However, state-of-
the-art bare-metal tools fail to provide richness and completeness in the
results of the analysis. In this context, Dynamic Binary Instrumentation
(DBI) tools have become popular in the analysis of new malware samples
because of the deep control they guarantee over the instrumented binary.
As a consequence, malware authors developed new techniques, called *anti-
instrumentation*, aimed at detecting if a sample is being instrumented. We
propose a practical approach to make DBI frameworks more stealthy and
resilient against anti-instrumentation attacks. We studied the common
techniques used by malware to detect the presence of a DBI tool, and we
proposed a set of countermeasures to address them. We implemented our
approach in ARANCINO, on top of the Intel Pin framework. Armed with it,
we perform the first large-scale measurement of the anti-instrumentation
techniques employed by modern malware. Finally, we leveraged our tool
to implement a generic unpacker, showing some case studies of the anti-
instrumentation techniques used by known packers.

## 1   Introduction

Malware is still one of the Internet's major security threat and dynamic analysis
systems are an essential component to a defend from it. By running malicious sam-
ples in a controlled environment, security analysts can observe their behavior [30],
unpack obfuscated code [25], identify command and control (C&C) servers, gen-
erate behavioral signatures, as well as remediation procedures for infections [12].
However, modern malware samples present many techniques, called *anti-debugging*
and *anti-sandboxing*, to hide their behavior when they detect that they are running
in a controlled environment. Generally, such malware is known as "evasive".

Previous research proposed different mechanisms to defeat evasive malware,
mainly based on two approaches. A first approach draws upon techniques (com-
monly known as "multi-path execution") used to increase the coverage of dynamic
analysis tools [10, 28, 42]. Binaries are executed multiple times, either providing

different inputs that invert the outcomes of conditional branches (i.e., trying to disarm detection triggers) [10, 28], or simply forcing the execution along a different path [42]. The second approach tries to analyze malware samples in environments that minimize the artifacts that can be exploited to detect the analysis system, for instance running samples on bare-metal [20], or porting the instrumenting framework in hardware [37]. While the former approach is more generic, it is often unfeasible since it leads to an explosion in the need of computational resources. Instead, the latter is more practical and, in fact, it has been recently adopted by security researchers [20, 21, 37].

To evaluate a malware analysis system, one should consider three factors: quality, efficiency, and stealthiness [20]. Quality refers to the richness and completeness of the analysis results. Efficiency measures the number of samples that can be analyzed per unit time. Stealthiness refers to the undetectability of the presence of the analysis environment. Intuitively, there is a constant trade-off between these factors. Despite the effort of the research community in developing approaches and tools to enforce the stealthiness of malware analysis systems, state-of-the-art tools provide either stealthiness or quality. For instance, Kirat et al. [20] proposed a framework for performing analyses on bare-metal. However, such systems cannot guarantee a good quality, i.e., they do not provide the analysts a good level of details in the inspection of the malware behavior. Instead, in some specific scenarios (e.g., manual and automated reverse engineering) we need deeper control of the running binary. In fact, the use of Dynamic Binary Instrumentation (DBI) frameworks has become a valid alternative for analyzing malware samples [4, 16, 27, 26]. By injecting instrumentation code inside the binary under analysis, DBI tools give a deep control over the analyzed program, providing both a low-level (i.e., instruction) and a high-level (i.e., system call) view. Given that, in order to keep on hindering the reverse engineering of their code, malware authors developed a new series of techniques, called *anti-instrumentation*, aimed at detecting the presence of a DBI tool at runtime.

We propose a dynamic protection framework to defend a generic DBI Tool against anti-instrumentation attacks. First, we classified state-of-the-art techniques in four categories (code cache artifacts, environment artifacts, just in time compiler detection, and overhead detection); then, we designed a set of generic countermeasures to defeat every identified class. In order to achieve this, we leverage the power of DBI tools to fully control the execution flow of the instrumented process. This allow us to detect and dismantle possible evasion attempts. We implemented our approach in ARANCINO, on top of the Intel Pin framework [23].

We tested our system against eXait [15], a tool containing a set of plugins that aim at detecting when a program is instrumented by Intel Pin, showing that ARANCINO is able to hide Intel Pin, allowing the analysis of evasive binaries. Then, we used our tool to perform a large-scale study of the anti-instrumentation techniques used by malware: we collected and analyzed 7,006 malware samples, monitoring the evasive behaviors that triggered our system. Finally, to show a useful application scenario of our system, we leveraged ARANCINO to implement a generic, dynamic, unpacker that can handle anti-instrumentation-equipped

packers. Moreover, we show case studies of known packers that employ anti-instrumentation techniques in the unpacking process.

In summary, we make the following contributions:

- We proposed an approach to practically defeat the techniques that malware employs to detect instrumentation systems (i.e., DBI tools).
- We performed a study of the common techniques adopted by modern malware authors to perform evasion of instrumentation systems. We measured how malware detects DBI tools on a dataset of 7,006 samples.
- We leveraged our anti-evasion approach to implement a generic, dynamic, evasion-resilient unpacker, showing some case studies of the evasion techniques used by know packers.

In the spirit of open science, we make our datasets and the code developed for ARANCINO publicly available [1].

## 2   Background and Motivation

Dynamic Binary Instrumentation (DBI) is a method to analyze the behavior of a binary application at run-time by injecting instrumentation code, which executes transparently as part of the normal instruction stream after being injected. Commonly, DBI tools exploit a Just In Time (JIT) compiler to instrument the analyzed binary at run-time, translating originally x86 code into an intermediated representation. Translation units used by different frameworks differ in size. For instance, Valgrind [29] uses standard basic blocks as traces, caches the translation, and jumps inside the corresponding cache each time a basic block is hit. DynamoRIO [9], instead, uses an extended definition of basic block as trace, including all instructions until a conditional jump. This includes also code inside function calls or after an unconditional jump. Intel Pin, uses traces defined as all the code between the previous trace and the next unconditional jump.

DBI tools turned out to be particularly useful in malware analysis, for instance to detect malicious software [4], identify cryptographic primitives [16], or to efficiently perform taint analysis [26]. However, DBI tools are not completely transparent to the analyzed malware, and, in fact, anti-instrumentation techniques have been developed to detect the instrumentation process. We classified such anti-instrumentation techniques in four categories.

**Code Cache Artifacts.** These techniques aim at detecting artifacts that are inherent of a DBI cache. For example, the Extended Instruction Pointer (`EIP`) is different if a binary is instrumented. In fact, in a DBI Tool the code is executed from a different memory region, called *code cache*, rather than from the main module of the binary.

**Environment Artifacts.** The memory layout of an instrumented binary is deeply different respect to the one of a not instrumented one. Searching for DBI artifacts such as strings or particular code patterns in memory can eventually

---

[1] `https://github.com/necst/arancino`

reveal the presence of a DBI tool inside the target process memory. Also, the parent process of an instrumented binary is often the DBI tool itself.

**JIT Compiler Detection.** JIT compilers make a lot of noise inside the process in terms of Windows API calls and pages allocation. These artifacts can be leveraged by the instrumented program to detect the presence of a DBI tool.

**Overhead Detection.** Instrumentation adds an observable overhead in the execution of a target program. This overhead can be noticed by malware by estimating the execution time of a particular set of instructions.

## 3  Arancino: Approach

A DBI tool can work at three granularities: (1) Instruction, a single instruction inside a collected trace; (2) Basic block, a sequence of instructions terminated by a conditional jump; (3) Trace, a sequence of basic blocks terminated by an unconditional jump. Our approach leverages the complete control that a DBI tool has on the instrumented binary to hide the artifacts that the DBI tool itself introduces during the instrumentation process. In fact, by instrumenting a binary, we can identify when it tries to leverage such artifacts to evade the analysis. In practice, we designed a set of countermeasures for the anti-instrumentation techniques we mentioned in Section 2. We implemented our approach in Arancino using Intel Pin. While we developed our countermeasures specifically targeting Intel Pin, the approach on which such countermeasures are based is generic and can be adapted to any DBI tool.

### 3.1  Code Cache Artifacts

DBI tools do not execute the original code of a program. Instead, they execute a particular memory region called code cache in which they copied an instrumented version of the original code.

**EIP Detection.** A malware sample can detect if the Instruction Pointer (`EIP`) is different from the expected one. This technique exploits particular instructions that save the execution context, including the current value of the `EIP` register. Once obtained the effective value inside `EIP` register, this value is compared with the base address in which the program expects the Instruction Pointer to be (main module of the program). If these two values are different, the program under analysis become aware that it has been instrumented. Here we report the currently working methods we identified to retrieve `EIP`. Examples of such instructions are `int 2e`, `fsave`, and `fxsave`. An easier technique to retrieve the `EIP` is monitoring the stack after a `call` instruction. However, this scenario is already inherently handled by Intel Pin, which does not alter the original `EIP` value on the stack. In order to defeat this technique, since there are a set of instructions that expose the presence of the DBI by leaking the effective `EIP`, we track these instructions in the collected trace and block the information leakage. Specifically, when we detect such instructions we execute a user defined function

that patches the value inside registers or memory with the expected one, so that the analyzed program can be fooled.

`int 2e` is the legacy instruction used in Windows in order to perform a system call (now replaced by `sysenter`). The interrupt handler called as consequence of the `int 2e` saves the current state information parameters, such as the EIP and the EFLAGS, in order to eventually return correctly from kernel-mode to user-mode after the execution of the syscall. Practically, this behavior has the side effect to store the EIP in the EDX register. When we detect the presence of `int 2e`, we insert a call to a function right after its execution that patches the value stored in EDX with the real EIP of the current instruction inside the main module.

`fsave`/ `fxsave`/ `fstenv` instructions save the floating-point context, including the effective EIP, at the moment of the execution of the last floating-point operation. When we detect one of these instructions in a trace, we insert a call to a function that manipulates the floating-point context by modifying the value of EIP with the correct address inside the main module of the program.

**Self-Modifying Code.** Another way to detect the presence of a code cache is using self-modifying code, i.e., code that changes the instructions at run-time by overwriting itself. Because a DBI tool executes the cache where the code is not modified, it will execute a wrong trace breaking the semantic of the program. This violates the property of semantic equivalence between the original program and the instrumented one. To handle this technique, we force the DBI tool to build a new trace with the correctly patched code and abort the execution of the wrong one. When a new trace is collected, we first retrieve its boundaries, then for each write instruction we check if the written address is inside the boundaries of the current trace. If this happens, the address is marked as *written*. At the same time, for each instruction we check if the EIP that has to be executed is marked as *written* and, if so, we force the DBI tool to discard the trace and build a new one.

## 3.2   Environment Artifacts

DBI tools introduce further environment artifacts during instrumentation.

**Parent Detection.** Since the instrumented program is launched by the DBI tool as its child, it can obtain its parent process and check if it is not the expected one. In order to hide the parent process name, we took into account the possibility to get the parent process not only by using the standard Windows APIs, but also using more advanced methods. Analyzing the various techniques aimed to retrieve the process list we have identified two countermeasures that can correctly defeat this type of attacks. The first one consists in hooking the `NtQuerySystemInformation` and faking the returned `SYSTEM PROCESS INFO` structure by replacing every process named `pin.exe` with `cmd.exe`. Second, we deny the instrumented program to open the process `CSRSS.exe`. In fact, this process is responsible to maintain its own process list in user-space. To do so, we create a list of PID of processes that we want to hide from the target program, then we hook the `NtOpenProcess` to check if the target PID is included in the list. If so, we trigger an `NTSTATUS_ACCESS_DENIED`.

**Memory Fingerprinting.** Malware can identify the presence of several artifacts that a DBI tool unavoidably leaves in memory. The most straightforward technique is to scan the entire memory of the process looking for allocated pages and searching inside these pages for a set of DBI-related artifacts (like strings and code patterns). This kind of attack is thwarted by hiding all the memory regions where the DBI-related code and data reside. This is done by intercepting and controlling the results of the functions ( `VirtualQuery`, `NtQueryVirtualMemory`) used to determine the state of a particular memory page. We check if the queried memory is inside a whitelist of addresses that the instrumented process is authorized to access. In the case the address is in the whitelist, we return the correct value, otherwise we return a `MEM_FREE` value. In other words, the memory space inside the whitelist is the only part that the analyzed program can see as allocated.

The whitelist is created at the beginning of the instrumentation process and updated at run-time when we detect a new dynamic memory allocation made by the analyzed program. Initially, the whitelist contains the main module of the executable. Then, when we detect that a new library is being loaded, we update the whitelist including such memory region. The heap and stack memory space is also whitelisted. We use `getProcessHeaps` at start-up in order to add to the whitelist the heap memory addresses created during the initialization process. We hook heap allocation functions to keep track of the dynamically allocated contents. By hooking the `NtAllocateVirtualMemory`, `RtlAllocateHeap`, and `RtlReAllocateHeap` functions we can actually cover all allocation functions since all the higher level APIs end up using one of them. Instead, to identify the stack we recover the address from `ESP` register and, using the default stack size (1048576 bytes), we whitelist its memory address space. Moreover, we need to add to our whitelist the PEB and TEB data structures, which contain information about the current process and threads. While analyzing these structures we also whitelist different memory regions pointed by fields in these structures. Last, we whitelist mapped files. Mapped files are particular memory structures in which the content of a file is associated with a portion of the process memory space. Some memory mapped files are created during the loading process of the executable, e.g., `locale.nls`, while others are created at run-time using the `MapViewOfFile` function, which at a lower level invokes the `NtMapViewOfSection` system call. Therefore, we scan the memory of the process as soon as it is loaded whitelisting all the memory regions marked as `MEM_MAPPED`. This allows us to consider the memory mapped files which are created before we get actual control of the execution. Moreover. we hook the `NtMapViewOfSection` system call to track the mapped files which are created by the application itself at run-time.

Note that a sample could detect our system by trying to allocate memory that is already allocated but hidden by our system. To this end, malware should allocate and deallocate through the entire address space to identify such memory regions. However, this is inefficient for the malware, and ARANCINO could easily detect such weird behavior.

### 3.3 JIT Compiler Detection

As briefly explained in Section 2, DBI tools use JIT compilers to instrument a binary. Malware can leverage JIT artifacts to spot the presence of a DBI tool.

**DLL Hook.** DBI tools need to hook functions in order to intercept the execution at the end of a not instrumented trace. Specifically, they insert jumps (Hooks) at the beginning of some functions. Consequently, an instrumented program can check if the initial instructions of common functions have been modified. In order to hide these differences, at the beginning of the instrumentation we retrieve the addresses in memory of the hooked functions (for example using the Windows API `GetModuleHandle`). Using this method, we can correctly retrieve addresses even if Address Space Layout Randomization (ASLR) is active on the machine. Once we know these addresses, we label them as *protected*, and we store the original values in a different memory region. When the binary wants to read a protected memory region, we simply redirect the read to our fake pages. In this way, the instrumented binary obtains the original value.

**Memory Page Permissions.** A JIT compiler needs to write and then execute a significant amount of code, hence the number of pages marked as RWX inside the address space is substantially larger respect to the not-instrumented execution of the program. A binary can scan the entire process address space counting the number of page encountered with RWX permissions. If this number is considerably high, it can spot the presence of the DBI. To address this technique, we leverage a whitelist of memory pages as we described in Section 3.2. When the instrumented binary performs a query of a memory region, we check if the address of such region is included in the whitelist of authorized addresses. If not, we return a `MEM_FREE` value, hiding the pages used by the JIT compiler.

**Memory Allocations.** As we mention before, the JIT compiler intensely uses RWX memory pages to instrument and cache instrumented traces. To this end, the DBI tool allocate the memory to fulfill the need. All these allocations are executed going through the low-level Windows API `ZwAllocateVirtualMemory`. A detection mechanism is to count the number of invocations of the `ZwAllocateVirtualMemory` and compare it with the number of invocations performed during a non-instrumented execution; if the difference is considerably high, the program can spot the presence of the JIT compiler. Usually, a program that wants to implement this anti-instrumentation technique inserts a simple assembly routine at the beginning of the `ZwAllocateVirtualMemory` which increments a counter every time the function is invoked; if the counter exceeds a predefined threshold then the JIT compiler is detected. The approach we used to defeat this technique aims at (1) redirecting every write attempt of the process inside the `.text` segment of libraries, and (2) faking the results of suspicious read instructions that have their target addresses inside the `.text` segment of libraries. This (1) prevents the insertion of hooks (e.g., in `ZwAllocateVirtualMemory`), and (2) denies the possibility to check if an hook is in place or not. Note that this approach is generic and can defeat every attempt to overwrite the original code of functions and to insert hooks inside protected memory regions.

### 3.4 Overhead Detection.

Despite DBI tools' effort to reach good instrumentation performance, they introduce an observable overhead, which can be detected by performing small benchmarks tests. To do so, malware leverages Windows APIs such as `NtQueryPerformanceCounter`, `GetTickCount`, `timeGetTime` to obtain execution times and reveal the presence of instrumentation tools. Our countermeasure tries to fool the process to think that the elapsed time is less despite the introduced overhead. We can achieve this by lowering the results returned from the previously described functions.

**Windows Time.** Windows APIs, such as `GetTickCount` and `timeGetTime`, retrieve the time information by accessing different fields in a shared user page mapped at the end of the process memory space. In this memory area, we can find the structure `KUSER_SHARED_DATA`. Our approach is to implement a "FakeMemory" in the locations that these functions access, lying about its content to cause the functions to return controlled time values. Hooking the API is not enough, because it is possible to read such data structure directly from the memory without calling any API. The strategy adopted in order to return fake values varies for the `GetTickCount` and the `timeGetTime`, because in order to calculate the elapsed time they employ two different methods.

`GetTickCount` accesses the fields `TickCountMultiplier` and `TickCountQuad` and performs a computation that returns the number of milliseconds elapsed since the start of the system. Our approach is to intercept the read of the `TickCountMultiplier`, retrieve the real value, divide it by a user defined `TICK DIVISOR` and finally return this value to the read instruction. This approach relies on a user defined divisor that must be tuned in order to successfully defeat this technique. `timeGetTime` accesses the field `interrupt_time` that is represented by a struct called `_KSYSTEM_TIME`. The time is retrieved by concatenating two fields of this struct: the `High1Time` and the `LowPart` as `High1Time:LowPart`. We reassemble the value, divide it by the user defined divisor, split it again in high and low part, and finally return to the read instruction the part that it was reading.

The second way that can be followed in order to retrieve the time elapsed is to employ the `QueryPerformanceCounter`. Since it is only a wrapper of `NtQueryPerformanceCounter`, we hooked directly the system call. Following this strategy after the execution we divide the value of the field `QuadPart` (a signed 64-bit integer) inside the `LARGE_INTEGER` struct returned by the syscall, with a user defined constant.

**CPU Time.** The last technique that we consider in order to defeat the time attack is the use of the assembly instruction `rdtsc`. This assembly instruction returns the processor timestamp defined as the number of clock cycles elapsed since the last reset. Since this is a 64-bit integer it returns, in a i386 architecture, the high part and the low part respectively in `EDX` and `EAX` registers. In order to fake the value returned from this instruction we recognize the `rdtsc` in a trace, then after its execution, we insert a call to a function that compose the 64-bit

Table 1: Top 20 malware families in our dataset.

| Family | No. Samples | No. Evasive | No. Techniques |
|---|---|---|---|
| virlock | 619 (8.8%) | 600 (96.9%) | 2 |
| confidence | 505 (7.2%) | 68 (13.5%) | 4 |
| virut | 242 (3.5%) | 13 (5.4%) | 2 |
| mira | 230 (3.3%) | 9 (3.9%) | 1 |
| upatre | 187 (2.7%) | 2 (1.1%) | 1 |
| lamer | 171 (2.4%) | 0 (0.0%) | 0 |
| sivis | 168 (2.4%) | 0 (0.0%) | 0 |
| installcore | 164 (2.3%) | 0 (0.0%) | 0 |
| ipamor | 164 (2.3%) | 0 (0.0%) | 0 |
| vtflooder | 152 (2.2%) | 2 (1.3%) | 1 |
| downloadguide | 135 (1.9%) | 1 (0.7%) | 1 |
| bladabindi | 103 (1.5%) | 22 (21.4%) | 2 |
| dealply | 98 (1.4%) | 1 (1.0%) | 1 |
| mydoom | 88 (1.3%) | 0 (0.0%) | 0 |
| parite | 86 (1.2%) | 18 (20.9%) | 1 |
| zusy | 84 (1.2%) | 10 (11.9%) | 3 |
| installmonster | 79 (1.1%) | 2 (2.5%) | 1 |
| allaple | 68 (1.0%) | 0 (0.0%) | 0 |
| razy | 66 (0.9%) | 10 (15.2%) | 3 |
| neshta | 62 (0.9%) | 0 (0.0%) | 0 |
| Others | 3,535 | 335 | 4 |
| *Total* | 7,006 | 1,093 (15.6%) | 5 |

integer as specified (`EDX:EAX`), divide it by a user defined constant, and patch the value of the two registers according to their new value.

## 4 Large-scale Anti-Instrumentation Measurement

We performed a study to measure which anti-instrumentation techniques are used by recent, malware samples found in the wild. To do this, we refer to the techniques that we described in the previous sections.

**Environment Setup.** We prepared a set of identical VirtualBox virtual machines running Windows 7 (64-bit), in which we installed common utilities such as Adobe Reader, alternative Web browsers, and media players. We also included typical user data such as saved credentials and browser history and, at runtime, our analysis system emulates basic user activity (e.g., moving the mouse, launching applications). This is useful to create a legitimate-looking environment. As suggested by Rossow et al. [32], we followed the best practices for malware experiments. We let the malware samples run for 5 minutes allowing samples to communicate with their control servers, and denying any potentially harmful traffic (e.g., spam). We automated our analysis environment to manage and control such VMs. For each analysis, our agent inside the VM receives the sample to be analyzed and executes it, instrumenting it with ARANCINO. After each execution, we save the logs produced by ARANCINO and roll back each virtual machine to a clean snapshot.

**Dataset.** Between October 2016 and February 2017, we used the VirusTotal Intelligence API to obtain the most recent Windows executables labeled as malicious by at least 3 AVs. We obtained a dataset of 7,006 samples. We then leveraged

Table 2: Top 10 most evasive malware families in our dataset. We considered only the families with at least 10 samples.

| Family | No. Samples | No. Evasive | No. Techniques |
|---|---|---|---|
| sfone | 19 | 19 (100.0%) | 1 |
| unruy | 11 | 11 (100.0%) | 1 |
| virlock | 619 | 600 (96.9%) | 2 |
| vilsel | 13 | 8 (61.5%) | 2 |
| urelas | 19 | 9 (47.4%) | 2 |
| confuser | 18 | 8 (44.4%) | 1 |
| vobfus | 52 | 19 (36.5%) | 1 |
| swisyn | 29 | 10 (34.5%) | 1 |
| softcnapp | 17 | 5 (29.4%) | 1 |
| downloadsponsor | 24 | 7 (29.2%) | 1 |

Table 3: Anti-instrumentation techniques used by malware in our dataset.

| Category | Technique | No. Samples |
|---|---|---|
| Code Cache Artifacts | Self-modifying code | 897 |
| Environment Artifacts | Parent detection | 259 |
| JIT Compiler Detection | Write on protected memory region | 40 |
| Environment Artifacts | Leak DEBUG flag | 5 |
| Environment Artifacts | Memory fingerprinting | 3 |

AVClass [34] to cluster AV labels and identify each sample's family. Table 1 shows the top 20 malware families in our dataset.

**Results.** We found out that 1,093 (15.6%) samples presented at least one anti-instrumentation technique. In particular, as shown in Table 3, we identified malware employing 5 different techniques, with the majority of the samples exposing self-modifying code capabilities. Table 2 shows the 10 most evasive malware families in our dataset. We leveraged AVClass [34] to determine the most likely family of each sample, and then we ranked families with more than 10 samples by the percentage of samples showing at least one anti-instrumentation technique. Interestingly, from Tables 1 and 3 we can see that, in some cases, not all the samples of a given family presented anti-instrumentation capabilities. This might mean that, at a certain point, malware authors decided to update their samples providing them with such features.

As further discussed in Section 7, we cannot assure that some of the detected anti-instrumentation techniques were added for the precise purpose of DBI evasion (instead of generic evasion techniques). Intuitively, this is dependent from the specific technique. For instance, while the detection of parent process could be performed by malware for different reasons (e.g., detecting a debugger or a specific agent inside a known sandbox) other than spotting the presence of Intel Pin, detecting writes and reads on protected memory regions is more specific to DBI-evasion.

From our study, we conclude that ant-instrumentation techniques, are employed by a significant amount of malware samples, motivating the effort in enforcing DBI tools and showing that systems such as ARANCINO can be useful in defeating these evasion techniques.

# 5 Application Scenario: Unpacker

On top of ARANCINO, we implemented a generic, anti-instrumentation-resilient unpacker. Our tool leverages the functionality provided by Intel Pin to track, with an instruction level granularity, the memory addresses that are written and then executed. As explained in Section 5.1, first of all, we identify a subset of instructions that are relevant for our analysis. Then, we keep track of each written memory region in order to create a list of contiguous written memory ranges, defined as *write intervals*. At the same time, we check if each executed instruction belongs to a write interval —this is the typical behavior of a packed binary that is executing the unpacked layer. If so, we trigger a further stage that consists of: (1) Dumping the main image of the PE and a memory range of the heap; (2) Reconstructing the Import Address Table (IAT) and generating the correct Import Directory; (3) Applying a set of heuristics (entropy, long jump, jump outer section, pushad popad, init function calls) to evaluate if the current instruction is the Original Entry Point (OEP). The result of our tool is a set of memory dumps or reconstructed PEs (depending on the success of the IAT fixing phase), and a report which includes the values of each heuristic for every dump. Based on those heuristics we can choose the best dump.

## 5.1 Unpacking Phases

In this section, we describe more in detail our unpacking approach.

**Instructions Filtering.** Since our tool works with an instruction level granularity, limiting our analysis to the relevant instructions of the program is critical. For this reason, we have introduced some filters, based on the common behaviors showed by packers. Specifically, we do not track two kinds of instructions: (1) write instruction on the Thread Environment Block (TEB) and on the stack, and (2) instruction executed by known Windows libraries.

The write instructions on the stack are ignored because unpacking code on the stack is not common. The same consideration can be applied to the instructions that write on the TEB, since most of these writes are related to the creation of exception handlers. The instructions executed by known Windows libraries are never considered when checking if the current instruction address is contained inside a write interval because this would mean that the packer writes the malicious payload in the address space of a known Windows library. This behavior has never been identified in the analyzed packers; moreover, this approach would break the application if it explicitly uses one of the functions which have been overwritten.

**Written Addresses Tracing.** In order to correctly operate, most packers need to write the original program in memory and then execute it. We can detect this behavior by tracing each write operation performed by the instrumented program and annotating a set of memory ranges that identify contiguously written memory addresses.

**WxorX Addresses Notifier.** As expressed before, packers need to both write and execute a specific memory region. Indeed, this behavior is distant from how

common benign programs act during execution. Although some benign programs may present such behavior, e.g., JIT compiler, virtual machine, programming languages interpreters, we can exploit such behavior as an evidence of something that was unpacked. Therefore, the first time we detect that the instruction pointer is inside one of the traced memory ranges, we trigger the the analysis and dumping routines, and mark the memory range as *unpacked*. We call the property of having a memory region that is either written or executed *WxorX rule*.

**Dumping Process.** While many memory dumping tools only look at the *main module* of the target program, which includes sections containing code, data, and the resources used by the binary, in order to handle more unpackers, we need to collect also code that has been unpacked on dynamic memory regions such as the heap. If the packer's stub unpacks new code on the heap and then redirects the execution there, a naive dump strategy that only looks at the sections of the PE would miss the unpacked code. We correctly catch this behavior by using the *WxorX address notifier*, and including in the dump all the memory regions that contain the unpacked code.

**IAT Fixing and Import Directory Reconstruction.** We focus our attention on three different techniques. The first technique we have to deal with is called *IAT redirection*: instead of having the addresses of the APIs directly in the IAT, entries contain addresses that point to jump instructions to the real API functions. The second technique is known as *stolen API*: it copies N instructions of the API function code at the address pointed by the IAT entry and then inserts an unconditional absolute jump to the N+1 instruction to the real API function code. The last one is a generalization of the stolen API technique. It differs from stolen API function because multiple parts of the original API functions are copied and they are connected together by absolute jumps until the last one reaches the original API code (Figure 1).

Thanks to Intel Pin, our system can deal with all these techniques using static analysis. Indeed, for each IAT entry we statically follow the control flow counting the number of instructions different from jumps, those are instructions belonging to the real API function (green, yellow and orange instruction in Figure 1). We follow the control flow until the target of a jump is inside a memory region occupied by the DLL. When the target address and the instructions count are retrieved, the value of the current analyzed IAT entry is replaced with the difference between the target address and the instructions count.

**Heuristics Description.** We used a set of heuristics during the unpacking process to understand when the program reaches the Original Entry Point (OEP) and can be considered fully unpacked. Howewer, as demonstrated in [33], understanding if the unpacking process is finished is not a decidable problem. The heuristics we have collected come from different works and books [36], [24], and we use them to tag a taken dump. These tags are useful at the end of the work of our tool in order to understand if a taken dump and the associated reconstructed PE represent the original program or not (since often the number of taken dumps is more than one). Here we describe the heuristics that we used:
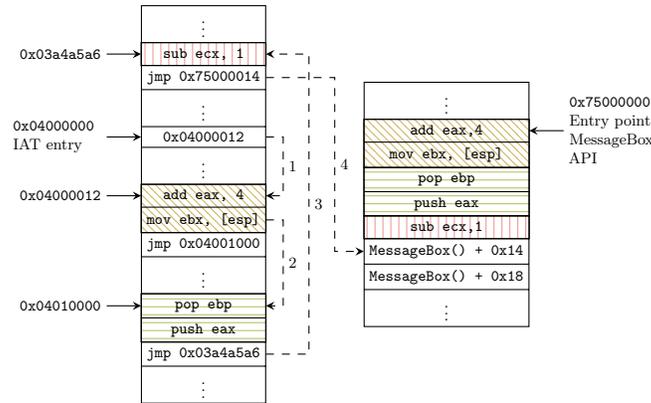
Figure 1: (1) The execution is redirected to the address inside the IAT entry. (2) The first two instructions of the original API are executed followed by a jump to the next chunk of instructions. (3) The chunk is executed together with another jump instruction. (4) The last instruction is executed and finally a jump goes inside the original memory space of the API.

- Entropy. We trace the entropy evolution of the main module of the target program and trigger a flag when the difference of the current entropy compared with the original one is greater than a threshold, as shown by Arora et al. [6].
- Jump outer section. A *long jump* is defined as a jump in which the offset between the target address and the previous EIP is larger than a threshold. This scenario identifies the typical behavior that transfers control from the packer's stub to the original program's code, or to another stub.
- Registers context switch. Usually stubs of packers show a particular behavior that is storing on the stack values of all the registers with a `pushad` instruction and, after the unpacking process has been completed, using a `popad` instruction to restore the previously saved values. This is done to maintain the correct context of the registers and avoid issues when the stub of a packer resumes the execution to the original code. Hence, the two instructions can be used to delimit the unpacking stub and help to identify the OEP.
- Suspicious imports. A packed binary usually includes very few imports, and reconstructs the correct IAT only at run-time. This is an obfuscation technique used by packers in order to hide the real behavior of the malware. The purpose of this heuristic is to search, in the reconstructed Import Directory, functions commonly used by the malware and not by the unpacking stub. For example, if a malware has to contact an Internet domain to download some malicious code, it has to have in its imports some Internet communication APIs, such as `connect` and `send`. Since the unpacking stub generally does not need them to perform its job, when the binary is packed, it does not have these API functions in its Import Directory initially. In some cases, the packed binary has as imports only the `GetProcAddress` and `LoadLibrary` functions, because these two are then used to dynamically load further functions.

# 6 Testing and Validation

## 6.1 Defeating Anti-instrumentation Techniques

We tested ARANCINO to evaluate its capabilities in defeating anti-instrumentation techniques. To do so, we leveraged eXait [15], a benchmark tool specifically designed to highlight different techniques aimed at spotting the presence of Intel Pin instrumentation. First, we instrumented and executed eXait with Intel Pin, then we did the same using ARANCINO. In both cases, we looked at the output of eXait to check if any of the techniques detected the instrumentation framework. Specifically, eXait implemented the following detection techniques: detecting common JIT API calls, detecting code and string artifacts, leaking the effective EIP, detecting by page permission, detecting ntdll hooks, detecting parent process, measuring overhead. While instrumented with Intel Pin, eXait flagged all of these techniques, spotting the presence of the DBI. Instead, when we instrumented it using ARANCINO, eXait did not produce any positive results, showing that our system successfully hid the presence of Intel Pin.

## 6.2 Unpacking Capabilities

**Testing Known Packers.** We tested our unpacker against 14 known packers, used to pack two binaries: a simple Message-Box (100KB) and WinRAR (2MB). We used different programs with different sizes because the behavior of a packer can be influenced by the size of the target program. Table 4 shows the effectiveness of our generic unpacking algorithm and PE reconstruction system against the different packers. For some packers, such as ASProtect, eXpressor and Obsidium we managed to take a dump at the correct OEP but, due to the presence of IAT obfuscation techniques, we were not able to reconstruct a working PE.

**Unpacking Malware Samples Found in the Wild.** We downloaded random samples from VirusTotal and we classified them with Exeinfo PE [1], a packer detection tool, in order to discard not packed samples. Eventually, we obtained a dataset of 1,066 packed samples, protected by known or custom packers, the latter tagged by Exeinfo PE as "unknown." Then, we tested the effectiveness of our unpacker against malicious samples spotted in the wild protected with both known and unknown packers. We automated the analysis of such samples using a VirtualBox VM and letting each sample, instrumented with our unpacker, run for 5 minutes. After the analysis performed by our tool, all the results were manually validated. Our unpacker managed to successfully unpack 808 (75.8%) samples, producing a working executable in 669 cases, while it failed to unpack the remaining 258 ones. The reasons of the failures can be different: evading the virtual environment, messing with the environment in a way that our script cannot manage to collect results, employing IAT obfuscation techniques, or exploiting packing techniques with a level of complexity out of our scope.

Table 4: Results of our tests against known packers. For each packer, here we show the packed program, if the original code of the binary has been successfully unpacked (U), if the reconstructed PE is a working executable (EX), the number of imports observed in the packed program, and the number of imports observed in the reconstructed PE.

| Packer | Binary | U | EX | Packed Imports | Recon/Tot Imports |
|---|---|---|---|---|---|
| Upx | MessageBox | ✓ | ✓ | 8 | 55/55 |
| FSG | MessageBox | ✓ | ✓ | 2 | 55/55 |
| Mew | MessageBox | ✓ | ✓ | 2 | 55/55 |
| Mpresss | MessageBox | ✓ | ✓ | 2 | 55/55 |
| Obsidium | MessageBox | ✓ | ✗ | 4 | 4/55 |
| PECompact | MessageBox | ✓ | ✓ | 4 | 55/55 |
| EXEpacker | MessageBox | ✓ | ✓ | 8 | 55/55 |
| WinUpack | MessageBox | ✓ | ✓ | 2 | 55/55 |
| ezip | MessageBox | ✓ | ✓ | 4 | 55/55 |
| Xcomp | MessageBox | ✓ | ✓ | 5 | 55/55 |
| PElock | MessageBox | ✓ | ✗ | 2 | 3/55 |
| Asprotect | MessageBox | ✓ | ✗ | 7 | 46/55 |
| Aspack | MessageBox | ✓ | ✓ | 3 | 55/55 |
| Hyperion | MessageBox | ✓ | ✓ | 2 | 55/55 |
| Upx | WinRAR | ✓ | ✓ | 16 | 433/433 |
| FSG | WinRAR | ✓ | ✓ | 2 | 433/433 |
| Mew | WinRAR | ✓ | ✓ | 2 | 433/433 |
| Mpress | WinRAR | ✓ | ✓ | 12 | 433/433 |
| Obsidium | WinRAR | ✓ | ✗ | 2 | 0/433 |
| PEcompact | WinRAR | ✓ | ✓ | 14 | 433/433 |
| EXEpacker | WinRAR | ✓ | ✓ | 16 | 433/433 |
| WinUpack | WinRAR | ✓ | ✓ | 2 | 433/433 |
| ezip | WinRAR | ✓ | ✓ | 12 | 433/433 |
| Xcomp | WinRAR | ✓ | ✓ | 5 | 433/433 |
| PElock | WinRAR | ✓ | ✗ | 2 | 71/433 |
| Asprotect | WinRAR | ✓ | ✓ | 16 | 433/433 |
| Aspack | WinRAR | ✓ | ✓ | 13 | 433/433 |
| Hyperion | WinRAR | ✓ | ✓ | 2 | 433/433 |

### 6.3 Case studies

In this Section, we present two case studies that show how known packers employ anti-instrumentation capabilities, and how ARANCINO successfully managed to overcome these techniques and instrument the binaries.

**Obsidium.** We found that Obsidium [2] performs a call to the function `QueryInformationProcess` using as `ProcessInformationClass` argument an undocumented class called `ProcessDebugFlags`. When `QueryInformationProcess` is called with this parameter, it returns the inverse of the field `EPROCESS.NoDebugInherit`, i.e., `TRUE` if the process is debugged, `FALSE` otherwise. Since this flag is set when a program is instrumented with Intel Pin, this attack can successfully reveal the presence of the DBI in memory. Instead, using ARANCINO, we managed to defeat this anti-instrumentation technique. In fact, by hooking the `NtQueryInformationProcess` system call and patching the `ProcessInformation` struct, as described in Section 3, ARANCINO correctly managed to instrument and execute a test binary. As result, our dynamic unpacker successfully unpacked samples packed with Obsidium.

Table 5: Overhead introduced by ARANCINO to defeat each technique.

| Technique | Execution Time [s] | Instrumentation Time [s] | ARANCINO Time [s] | ARANCINO Overhead [%] |
|---|---|---|---|---|
| EIP Detection - int2e | 0.01 | 0.71 | 1.15 | 61% |
| EIP Detection - fsave | 0.01 | 0.90 | 1.20 | 33% |
| EIP Detection - fxsave | 0.01 | 0.90 | 1.15 | 27% |
| EIP Detection - fstenv | 0.01 | 0.82 | 1.05 | 28% |
| Memory Page Permissions | 0.01 | 0.82 | 0.90 | 9% |
| DLL Hook | 0.01 | 0.81 | 2.00 | 145% |
| Memory Allocations | 0.01 | 2.00 | 2.90 | 45% |
| Windows Time | 0.01 | 2.93 | 5.51 | 88% |
| Parent Detection | 0.02 | 0.85 | 0.87 | 2% |
| Memory Fingerprinting | 0.04 | 2.00 | 7.09 | 254.5% |

**PEspin.** We found that PEspin [3] makes use of self-modifying code. In fact, when we instrumented a test binary packed with PEspin without ARANCINO, the program crashed. Instead, when using our tool, the execution reached its normal end flawlessly. To understand precisely the reason of the crash, we manually analyzed the packed binary. We ran it in a debugger and we identified two instructions, IN and OUT that provide a direct access to I/O devices and require to be executed in protected mode. As a consequence, a program running in user space, which is not allowed to do it, crashes. During normal executions, these instructions are overwritten before EIP reaches them, so they are never executed and they never cause the crash of the program. However, when the binary is instrumented, Intel Pin places a trace composed by the two instructions in the code cache, eventually running them. Instead, as explained in Section 3.1, ARANCINO is able to detect if instructions are modified, and discard the current trace forcing Intel Pin to collect a new one. This avoids the crash of the program.

### 6.4 Performance

The efforts made to improve the transparency of Intel Pin come with a performance cost. We measured three different times. First, the execution time of the program when not instrumented, used as reference to estimate the final overhead. Second, execution time of the instrumented executable without ARANCINO, used to evaluate the overhead introduced by Intel Pin alone. Finally, the execution time of the executable instrumented with ARANCINO, used to evaluate the overhead introduced by our tool. As shown in Table 5 the overhead introduced by our countermeasures is strictly dependent from the specific technique. For instance, while handling `int2e`, `fsave`, `fxsave`, `fstenv` instructions, ARANCINO introduces a low overhead. On the other hand, when we defeat detection of DLL hooks or memory fingerprinting, the overhead introduced by ARANCINO is high, because ARANCINO has to analyze each read instruction to find its destination address and check if it belongs to the whitelist.

## 7 Discussion of Limitations

Our work presents some limitation that we describe in the following paragraphs.

**Arancino's Artifacts.** Even if Arancino correctly hides a vast number of DBI artifacts, we cannot guarantee that it is completely immune to detection attempts. In fact, some of the anti-evasion techniques that we proposed might be leveraged as detection criteria. For instance, malware could exploit that fact that Arancino hides all the protected memory pages. Specifically, a malicious sample could try to allocate memory through the entire address space failing to access memory regions that are already allocated for those hidden pages. This does not tell the analyzed sample which libraries are loaded or the reason the OS is denying memory allocation, but it is a particular behavior that could be fingerprinted. Another example is the denied access to `CSRSS.exe`. On one side this prevents to know the list of running processes, on the other side this could be detected by a malware sample. However, although these limitations are present in the current implementation of Arancino, it is possible to implement more sophisticated mechanisms to hide such artifacts (e.g., allowing access to `CSRSS.exe` and altering its internal data structures) and partially overcome these limitations. In summary, we believe that Arancino definitely raises the bar for the attacker, making DBI evasion much harder.

**Anti-Instrumentation vs. General Artifacts.** The techniques we observed to identify DBI-evasion attempts are indicators of potential anti-instrumentation attacks, but they do not capture the intention of the malware authors. For instance, a self-modifying behavior, even if it breaks the use of code cache, may be used as an obfuscation technique as well as an anti-instrumentation feature. Similarly, the detection of parent process could be performed for different reasons (e.g., detecting agents of a known sandbox) other than spotting the presence of a DBI tool. We can only speculate why malware authors are adopting the techniques that we identified in our study. Independently from the intention of malware authors, our experiments showed that Arancino is able to hide many artifacts that malware uses to evade analysis systems.

**Measurements.** We based our approach and our study on the known techniques employed to detect instrumented environment. While we cannot assure that the current version of Arancino covers all the possible techniques used to detect the presence of a DBI tool, our approach and our tool can be easily extended in order to add support for new ones.

**Environment Setup.** We performed our study running malware on a virtualized environment (i.e., VirtualBox). While we setup our VMs to minimize the presence of VirtualBox artifacts, sophisticated malware samples can detect they are running in a virtualized environment and hide their malicious behavior. However, this is not a fundamental limitation of our approach, in fact, Arancino can perfectly work on top of bare-metal system such as [20] [21] [37] improving the quality of such systems without affecting the stealthiness.

**Implementation.** Arancino currently supports only 32-bit binaries. Extending Arancino to 64-bit binaries can introduce a new attack vector that exploits the Windows 64-bit backward compatibility with 32-bit applications to spot the presence of a DBI, making it crash. In fact, in order to implement the backward compatibility Windows runs each application using two different code segments,

one for the execution in 32-bit compatibility mode and one for the 64-bit mode. Each segment has a flag indicating in which mode its instructions have to be interpreted. Particular instructions, such as far `jmp` or far `ret`, can be used to jump from one segment to the other. This behavior can cause the program to switch between the 32- and the 64-bit segments. Unfortunately, DBI tools cannot correctly handle this behavior and, consequently, they may disassemble and instructions wrongly or, in the worst case, they may be unable to disassemble them at all. While we leave the implementation of 64-bit support for future work, the majority of malware samples nowadays are 32-bit binaries.

**Performance.** As shown in Section 6.4, ARANCINO inevitably introduces an overhead on the execution of the instrumented binary. While this overhead is often acceptable, it is quite high when ARANCINO handles some particular techniques.

**Unpacker.** First, our unpacker does not handle the process hollowing techniques, so runtime unpacking that injects the payload inside new processes and/or make use of DLL injection techniques are not considered by our tool. Second, if a packer implements a custom IAT obfuscation technique, we cannot reconstruct a runnable PE. Third, it does not track unpacking on the stack, so if the original code of the program is written on the stack our tool cannot dump the correct memory region and consequently it cannot reconstruct a working PE. However, this technique is not commonly employed by known packers. Finally, referring to the packer taxonomy presented in [38], our unpacker can handle packers with a maximum level of complexity of 4. To handle more sophisticated packer, our approach could be combined with approaches such as [39], which is less generic and targets specifically complex packers.

## 8   Related Work

**DBI-Evasive Malware.** Despite the fact that it is theoretically impossible to build a completely transparent analysis tool, different works have been done regarding the development of stealthy instrumentation systems.

SPiKE [41] adopts an approach completely different from other DBI tools, such as Intel Pin or DynamoRIO, because it does not use a JIT compiler in order to instrument traces. Instead, it places a series of breakpoints in memory, called *drifters*, that, when are hit, trigger appropriate hooks, called *instruments*. The drifters are implemented following the technique explained in VAMPiRE [40], which guarantees a certain level of transparency and the impossibility of removal by the malware—they are placed in not accessible memory regions. While this work is stealthier than other DBI tools, it is also much less powerful than the aforementioned ones, because it offers very few APIs. Another work that relies on setting stealthy breakpoint is

SPIDER [14]. It duplicates the program in two different views: the code view, and the data view. The former can be modified by the instrumentation framework with the insertion of breakpoint and it is where the instructions are fetched, while the latter is where all the write and read operations made by the program are

redirected and it can be modified only by the program itself, ensuring transparency of the framework. If the instruction fetched and executed from the code view hits a breakpoint, then a user-defined hook is invoked. As the work described before, SPIDER is less powerful than other DBI tools, as it only works at function-level.

**Malware Unpacking.** Ugarte-Pedrero et al. [38] recently proposed a taxonomy to classify packers by their complexity. Generally, we can classify unpackers in three categories: static [13], dynamic and hybrid unpackers. Most of the recent proposed works are dynamic unpackers [17, 44, 43, 7, 31, 8, 18]. These approaches leverage the fact that packers need of unpack the code in memory at runtime to obtain the unpacked code.

Omniunpack [25], monitors memory page writes, exploiting a memory protection mechanism provided by the operating system, to access unpacked code pages and notify external tools when such pages are ready to be analyzed. Another attempt of exploiting operating system capabilities was made in Eureka [35]. This tool uses a kernel driver to trace system calls and identify the time when the analyzed sample is unpacked in memory. Similarly to our tool, Renovo [19] instruments a packed binary to trace writes in memory and execute until the instruction pointer fall inside written regions. Unfortunately, Renovo is easily detectable by anti-instrumentation techniques. Moreover, its output is a not working PE because it is does not try to recover the IAT table. Rambo [39] exploits a multi-path execution technique to trigger the unpacking of code regions. This is useful to defeat those packers who check the environment to understand if they are under analysis. Complementary to our approach, Rambo tries to solve this problem by forcing the execution of not executed code, while our unpacker exploits ARANCINO to hide artifacts.

Hybrid unpacking mixes some static analysis techniques with dynamics ones. The strategy for hybrid unpacking can generally follow two approaches. First, static analysis followed by dynamic analysis: An initial static analysis phase is performed in order to extract a model of how the execution will look like and, after that the program is executed, the static model built before is checked against the dynamically executed instructions [33]. Second, dynamic analysis followed by static analysis: This scheme relies on collecting traces of the execution of the packed program, and eventually extracting, by static analysis, the interface and the body of the unpacking function [11].

Our dynamic unpacker provides, respect to the existing solutions, a richer synergy of heuristics aimed to automatically detect a possible OEP. Following the taxonomy proposed in [38], our unpacker is able to handle packers up to `type 4`. These packers are multi-layer, cyclic and interleaved. This means that they use different layers with both forward and backward transitions. Moreover, our unpacking algorithm is more generic respect to solution such as [22] since we remove the assumption about the last written at page: We do not assume that the last written and executed page is the one that contains the OEP; in fact, as demonstrated in [5], this is not always necessarily true. Finally, while previous approaches aim only at unpacking obfuscated code, our tool is able, in most of the cases, to produce a working (i.e., runnable) executable.

## 9 Conclusions

Dynamic Binary Instrumentation (DBI) frameworks provide useful features in the analysis of malware samples. However, modern malware samples implement anti-instrumentation techniques to detect if they are being instrumented. We proposed a practical approach to enforce DBI frameworks and make them stealthier against anti-instrumentation attacks by studying the common techniques used by malware to detect their presence, and proposing a set of countermeasures to address such techniques. We implemented our approach in ARANCINO on top of Intel Pin. Then, we used ARANCINO to perform a measurement of the anti-instrumentation techniques employed by malware. We analyzed 7,006 recent malware samples showing that 15.6% of them employ at least one anti-instrumentation technique. Moreover, we leveraged our evasion-resilient approach to implement a generic unpacker showing interesting case studies of the anti-instrumentation techniques used by known packers. Combining ARANCINO with our unpacker we provided a tool that can successfully analyze malware that employs both anti-static-analysis and anti-instrumentation techniques.

## Acknowledgements

## References

[1] *Exeinfo PE.* `http://exeinfo.atwebpages.com/`.

[2] *Obsidium.* `https://www.obsidium.de/show/download/en`.

[3] *PESpin.* `http://www.pespin.com/`.

[4] N. Aaraj, A. Raghunathan, and N. K. Jha. Dynamic binary instrumentation-based framework for malware defense. In *Proc. of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[5] S. Arne and M. Alaeddine. *One packer to rule them all: Empirical identification, comparison and circumvention of current Antivirus detection techniques.* `https://www.blackhat.com/docs/us-14/materials/us-14-Mesbahi-One-Packer-To-Rule-Them-All-WP.pdf`.

[6] R. Arora, A. Singh, H. Pareek, and U. R. Edara. A heuristics-based static analysis approach for detecting packed pe binaries. *International Journal of Security and Its Applications*, 2013.

[7] P. Bania. Generic unpacking of self-modifying, aggressive, packed binary programs. *arXiv preprint arXiv:0905.4581*, 2009.

[8] BromiumLabs. *The Packer Attacker is a generic hidden code extractor for Windows malware.* `https://github.com/BromiumLabs/PackerAttacker`.

[9] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[10] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection.* 2008.

[11] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. Technical report, DTIC Document, 2009.

[12] A. Continella, A. Guagnelli, G. Zingaro, G. De Pasquale, A. Barenghi, S. Zanero, and F. Maggi. Shieldfs: a self-healing, ransomware-aware filesystem. In *Proc. of the Annual Conference on Computer Security Applications (ACSAC)*, 2016.

[13] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. Automatic static unpacking of malware binaries. In *Proc. of Working Conference on Reverse Engineering (WCRE)*. IEEE, 2009.

[14] Z. Deng, X. Zhang, and D. Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2013.

[15] F. Falcon and N. Riva. Dynamic binary instrumentation frameworks: I know you're there spying on me. In *Proc. of Reverse Engineering Conference*, 2012.

[16] F. Gröbert, C. Willems, and T. Holz. Automated identification of cryptographic primitives in binary programs. In *Proc. of International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2011.

[17] F. Guo, P. Ferrie, and T.-C. Chiueh. A study of the packer problem and its solutions. In *Proc. of International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2008.

[18] Hex-Rays. *Ida Universal Unpacker.* `https://www.hex-rays.com/products/ida/support/tutorials/unpack_pe/index.shtml`.

[19] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. 2007.

[20] D. Kirat, G. Vigna, and C. Kruegel. Barebox: efficient malware analysis on baremetal. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2011.

[21] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proc. of USENIX Security*, 2014.

[22] J. Lenoir. *Implementing Your Own Generic Unpacker.*

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*. ACM, 2005.

[24] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. In *Proc. of IEEE symposium on Security and Privacy (SP)*. IEEE, 2007.

[25] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2007.

[26] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. Taintpipe: Pipelined symbolic taint analysis. In *Proc. of USENIX Security*, 2015.

[27] J. Ming, D. Xu, L. Wang, and D. Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[28] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. of IEEE symposium on Security and Privacy (SP)*, 2007.

[29] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[30] M. Polino, A. Scorti, F. Maggi, and S. Zanero. Jackdaw: Towards automatic reverse engineering of large datasets of binaries. In *Proc. of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[31] D. Quist. *Circumventing software armoring techniques.* `https://www.blackhat.com/presentations/bh-usa-07/Quist_and_Valsmith/Presentation/bh-usa-07-quist_and_valsmith.pdf`.

[32] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Proc. of IEEE symposium on Security and Privacy (SP)*, 2012.

[33] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. 2006.

[34] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In *Proc. of International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2016.

[35] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee. Eureka: A framework for enabling static malware analysis. In *European Symposium on Research in Computer Security*, pages 481–500. Springer, 2008.

[36] M. Sikorski and A. Honig. *Practical Malware Analysis.* No Starch Press, 2012.

[37] C. Spensky, H. Hu, and K. Leach. Lo-phi: Low observable physical host instrumentation. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[38] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Sok: deep packer inspection: a longitudinal study of the complexity of run-time packers. In *Proc. of IEEE symposium on Security and Privacy (SP)*. IEEE, 2015.

[39] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. Rambo: Run-time packer analysis with multiple branch observation. In *Proc. of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 186–206. Springer, 2016.

[40] A. Vasudevan and R. Yerraballi. Stealth breakpoints. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2005.

[41] A. Vasudevan and R. Yerraballi. Spike: engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proc. of the 29th Australasian Computer Science Conference-Volume 48*. Australian Computer Society, Inc., 2006.

[42] J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Proc. of International Workshop on Recent Advances in Intrusion Detection (RAID)*, 2007.

[43] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *Proc. of IEEE symposium on Security and Privacy (SP)*. IEEE, 2015.

[44] S.-C. Yu and Y.-C. Li. A unpacking and reconstruction system-agunpacker. In *Proc. of International Symposium on Computer Network and Multimedia Technology,(CNMT)*. IEEE, 2009.