# RAMBO: Run-time packer Analysis with Multiple Branch Observation

Xabier Ugarte-Pedrero[1 2], Davide Balzarotti[3], Igor Santos[1], and Pablo G. Bringas[1]

[1] University of Deusto, Bilbao, Spain
{xabier.ugarte,isantos,pablo.garcia.bringas}@deusto.es
[2] Cisco Talos Security Intelligence and Research Group
xabipedr@cisco.com
[3] Eurecom, Sophia Antipolis, France
davide.balzarotti@eurecom.fr

**Abstract.** Run-time packing is a technique employed by malware authors in order to conceal (e.g., encrypt) malicious code and recover it at run-time. In particular, some run-time packers only decrypt individual regions of code on demand, re-encrypting them again when they are not running. This technique is known as shifting decode frames and it can greatly complicate malware analysis. The first solution that comes to mind to analyze these samples is to apply multi-path exploration to trigger the unpacking of all the code regions. Unfortunately, multi-path exploration is known to have several limitations, such as its limited scalability for the analysis of real-world binaries. In this paper, we propose a set of domain-specific optimizations and heuristics to guide multi-path exploration and improve its efficiency and reliability for unpacking binaries protected with shifting decode frames.

**Keywords:** Malware, unpacking, multi-path exploration

## 1 Introduction

Malware authors employ a large variety of techniques to conceal their code and make reverse engineering and automatic detection more difficult. One of these techniques is packing, which consists in encoding or encrypting the code and data in the binary and revealing them only at run-time.

Packers have been widely studied by researchers and, as a result, many generic unpacking techniques have been proposed in the literature. In particular, researchers have addressed this problem from different perspectives: (i) by making the analysis platform resilient to anti-analysis techniques [1], (ii) by tracing the execution of the binary at different granularity levels [2, 3], (iii) by adopting different heuristics to detect the original entry point of the binary [4], or by dumping the code at the appropriate moment [5], and (iv) by improving the efficiency of the unpacking process [6]. Although some of these approaches use static analysis techniques [7], the majority rely on the execution of the sample.

Nevertheless, there is a specific protection technique that takes advantage of an intrinsic limitation of dynamic analysis, i.e., the fact that it only explores a single execution path. *Shifting-decode-frames* or *partial code revelation* consists of unpacking the code on demand, just before its execution. These packers only reveal one code region at a time, decrypting only the code covered by a single execution path. In previous work [8], we classified this behavior at the highest level of complexity (with the exception of virtualization based packers). One of the most common and famous packers that employ this technique is Armadillo, which is widely used among malware writers.

These protection scheme is particularly effective in cases in which the sample employs anti-sandbox techniques to conditionally execute the payload, or when it is designed to communicate with external entities (e.g., a Command and Control Server). If the sample is executed inside an isolated environment or the server is unavailable, certain parts of its code will never be executed under a single-path dynamic execution engine. In both cases, a packer like Armadillo would not reveal the portions of the code that are not executed.

Therefore, the first solution that may come to mind to deal with these packers is to resort to some form of multi-path exploration. Several works [9–13] have studied multi-path exploration to improve coverage in dynamic analysis. While these works address some of the limitations of dynamic analysis, none of them has addressed the specific problems that may arise when adopting this technique for the generic unpacking of samples protected with shifting-decode-frames.

On the one hand, packers heavily rely on self-modifying code and obfuscated control flow, making very hard to automatically explore different execution paths. One of the major limitations of multi-path exploration is its computational overhead, making the approach almost infeasible for large-scale malware analysis. On the other hand, in our case we do not need to execute all possible paths, but only to guide the execution in a way to maximize the recovered code. Moreover, as the program does not need to continue once the code has been unpacked, the memory consistency is less of an issue in the unpacking problem. As a result, multi-path exploration of packed programs is still an open and interesting problem, that requires a new set of dedicated and custom techniques.

Peng et al. [13] proposed the application of a fully inconsistent multi-path exploration approach and applied their technique to improve the execution path coverage in malware, focusing in particular on environment sensitive malware. In this paper, however, we focus on the specific characteristics of the described packing technique. These particularities allow us to apply different optimizations and heuristics to multi-path exploration, improving the feasibility of this technique, especially for complex cases.

In particular, in this paper we want to answer two questions: *Is it possible to apply new optimizations to the classic multi-path exploration to efficiently uncover protected regions of code for packers using shifting-decode-frames?* And *is it possible to design new heuristics specific to the unpacking domain, that can guide the multi-path exploration and increase the recovery of the protected code?*

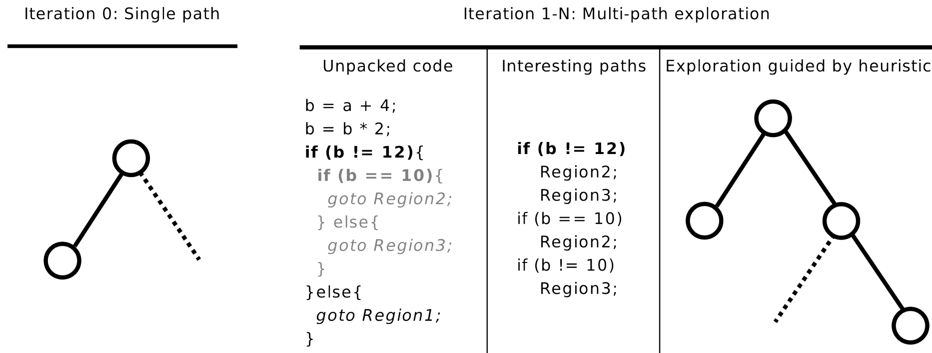


**Fig. 1.** General workflow of our approach.

Our main contributions are oriented to answer these questions: (i) we propose a set of optimizations for the application of multi-path exploration to binaries protected by shifting-decode-frames, (ii) we introduce a new heuristic that can guide multi-path exploration to unpack previously unseen regions of code and (iii) we evaluate this approach and present three different case studies.

## 2 Approach

Moser et al. [14] proposed for the first time the application of multi-path exploration for the analysis of environment-sensitive malware. This approach leveraged dynamic taint analysis, symbolic execution, and process snapshotting in order to explore multiple execution paths in depth-first order.

In order to evaluate our system, we implemented a modified version of multi-path exploration applying a set of domain specific optimizations that allow us to selectively explore certain *interesting* regions of code: which in our case is the code of the original program protected by the packer.

Our multi-path exploration engine is built on top of TEMU and Vine, the components of the Bitblaze [15] platform. TEMU allows to trace the execution of a binary, applying dynamic taint analysis, whereas Vine is an analysis engine based on Vine-IL, an intermediate language, that allows to design control-flow and data-flow analysis algorithms.

The general workflow of our solution is as follows (see Figure 1). We first execute the sample in a single-path execution mode and extract different pieces of information. We analyze the packer structure and identify the regions of memory that contain the protected code, by applying the techniques developed in previous work [8]. In a second step, we extract the memory that was unpacked in this first run, and compute the control flow graph of the unpacked code in order to find interesting points in the code (i.e., control flow instructions that lead to the unpacking of new regions of code). This process provides us a list containing the control flow instructions that lead to new regions. We use this list as part of a

heuristic to guide multi-path exploration. Finally, we apply our optimized multi-path exploration engine using this pre-computed information to prioritize paths that will likely drive to the unpacking of new regions. This two-step process is repeated until our system cannot recover any additional code.

This section is divided in three parts. First, we introduce our multi-path exploration approach, and describe several design decisions. Second, we describe a set of optimizations we developed over this model, and third, we present the heuristic that allows us to prioritize execution paths in this specific domain.

### 2.1 General Approach

**Symbolic execution.** Symbolic execution allows to evaluate a program over a set of symbolic inputs instead of concrete values. A constraint solver can evaluate the symbolic expression that must be satisfied to follow a given path, providing an appropriate set of values for each input variable. The reader can refer to previous literature [14, 16] for a better understanding of symbolic execution and its limitations.

Some symbolic execution engines [17, 18] simplify symbolic expressions to enhance the efficiency of the computations of the constraint solver. Alternatively, other works [19, 20] propose the use of weakest preconditions, a method that keeps the computational complexity and size of the formulas $O(n^2)$ [16]. We leveraged Vine, a tool that can compute the weakest precondition of an execution trace and to generate a query to the STP constraint solver.

Indirect memory accesses (i.e., memory access instructions in which the address itself is tainted and it depends on program input) are a recurrent problem in symbolic execution. When the program is evaluated symbolically, the address can contain any value constrained by the symbolic expression. This limitation is specially problematic for the symbolic execution of jump tables, a mechanism widely used by compilers to implement *switch* statements.

Some approaches let the constraint solver reason about the possible values, while other approaches perform alias analysis in order to determine the possible memory ranges pointed by the index [16]. In our case, we let Vine adopt the concrete value observed during the execution for every tainted memory index avoiding symbolic processing. Although this unsound assumption implies that some paths will never be executed, it simplifies the reasoning process involved in multi-path exploration. This limitation can be eventually mitigated in cases in which several paths in the program trigger the execution of a page or function, successfully triggering its unpacking routine.

**System-level snapshots.** In order to save the execution state at a given point (before a conditional jump is evaluated), we collect a system-snapshot. Previous approaches have proposed the use of process snapshots, a technique more efficient in terms of computational overhead and disk space. Nevertheless, making snapshots of the process state (memory and registers) involves many technical problems that are not easy to address. Processes running on the system generally use resources provided by the operating system like files, sockets, or the registry. Besides, the kernel of the operating system maintains many structures

with information regarding the memory assignment, heaps, stacks, threads, and handles. While saving and recovering the memory and register state is not difficult to implement, it is hard to maintain the system consistency when the state of a process is restored. Moser et al. [14] proposed several methods to ensure that the process can continue running even if it is restored to a previous state (e.g., avoiding closing handles).

Since the optimization of process snapshots is beyond the scope of this study and stands as a research problem by itself, we adopt a system-snapshot approach that, in spite of sacrificing system efficiency, allows us to securely restart the execution of a program at any point maintaining the consistency of the whole system.

**Taint sources.** We taint the output of the APIs that are most interesting for our goals, including network operations such as `connect`, `recv`, `gethostbyname` or `gethostbyaddr`, file operations such as `ReadFile` or `CreateFile`, command line argument related functions such as `__wgetmainargs` or `ReadConsoleInput`, and other functions typically used to query the system state like `GetSystemTime` or `Process32First` / `Process32Next`.

**Target Code Selection.** In *shifting-decode-frames*, we can distinguish two parts in the code. First, there must be a decryption routine that is usually highly obfuscated and armored with anti-analysis tricks. This routine is in charge of taking control when the execution of the protected code jumps from one region to another, decrypting the next region of code, and encrypting the previous one.

Our goal only requires to apply multi-path exploration to the protected code, avoiding the decryption and anti-analysis routines. In order to do this, we first need to determine the place where the original code is decrypted and executed. This problem has been widely studied in the past and researchers have proposed different heuristics. To this end, we implemented a framework based on a previous approach [8] to analyze the execution trace of the binary and divide the execution into layers. Our framework also incorporates several heuristics that can highlight the code sections that likely contain the original code. This information is also presented to the analyst who may select other regions to explore on demand, if necessary.

### 2.2 Domain specific optimizations

In this section we introduce six custom optimizations that simplify the multi-path exploration problem in the case of binary unpacking.

**Inconsistent multi-path exploration.** In some cases, traditional symbolic execution approaches cannot execute certain paths that, despite of being feasible, are difficult to solve for a constraint solver. For instance, a parser routine may access tables with a symbolic index. Reasoning about indirect symbolic memory accesses requires a complex processing such as alias analysis.

In these cases, when our constraint solver cannot provide a solution, we take an unsound assumption and query the constraint solver ignoring the path

restrictions imposed by the previous instructions in the trace. This approach lets us explore the path by forcing a set of values consistent with the last tainted jump instruction, but potentially inconsistent with the previous path restrictions.

In our specific domain, maintaining the consistency of the system is only important in order to avoid system crashes until every protected region of code has been unpacked. While other domains may suffer from this unsound implementation (e.g., malware analysis may require to know under which circumstances a certain path is triggered), in our case this information is not relevant, as long as the system remains stable enough to unpack the different regions.

**Partial symbolic execution.** In order to reduce the size of the code to be explored symbolically, we restrict symbolic execution to the original malicious code (i.e., unpacking routines are explored in single-path execution mode). One may think that the unpacking code will never have conditional branches that depend on system input, but there are packers, like Armadillo, that apart from protecting the original code of the binary, apply licensing restrictions. Moreover, this packer fetches the system date using the `GetSystemTime` API function in `kernel32.dll`, and executes conditional jumps that depend on the information collected. Nevertheless, this code will not trigger the unpacking of new regions of code. Also, this code is generally highly obfuscated and does not follow standard calling conventions, making more difficult to correctly trace and symbolically process this code. For these reasons, we restrict multi-path exploration to the regions suspected to contain the code of the original application.

**Local and global consistency.** Another aspect to consider is the consistency of the symbolic execution engine. For example, the S2E project [18] allows to run programs at different consistency levels.

In order to minimize the computational overhead we apply a locally consistent multi-path exploration approach. This means that we respect the consistency within the regions that contain the original code of the binary, but we allow the variables in this region to adopt values that are inconsistent with the rest of the code (e.g., system libraries). For instance, a program may update a variable with a value coming from keyboard input after a `scanf` call. This function applies some restrictions to the input, as well as some parsing. As a result, the value adopted by the variable would be restricted by the (potentially complex code) present in the library. In order to avoid this complexity, we let the variable adopt any value creating a fresh symbolic variable for it.

First, we avoid tracing any taint-propagating instruction if it is executed outside the explored regions. In this way, when the execution trace is processed in the symbolic engine, only the instructions in the explored regions impose restrictions over the symbolic variables.

Second, the first time a new taint (that has been created outside the interesting regions of code), is propagated to our explored code, we create a completely new taint value for each of the memory bytes affected by this taint, in such a way that our system will consider those bytes as free variables.

Finally, whenever the program calls to a function outside the region delimited, if the arguments of the call are tainted then the result of the call can

be consequently tainted. As we do not record the execution of such code, the taint propagation chain will be broken and our tool will be unable to provide a solution. Executing symbolically all the code present in these API functions can become computationally infeasible. For this reason, we avoid recording the execution of code outside the boundaries of our regions of interest. In order to allow Vine to process these traces with broken taint propagation, we create a new independent symbolic variable whenever necessary. In this case, again, we lose program consistency. Nevertheless, as we describe in Section 2.2, this inconsistency does not affect our approach but on the contrary, lets us explore as many paths as possible (triggering the execution and thus the unpacking of new regions of code).

**State Explosion.** One of the limitations that make multipath exploration infeasible to analyse large programs is the well-known state explosion problem [21]: when the number of state variables increases, the number of states grows exponentially. Many samples may have infinite program states, for example when unbounded loops are implemented in the explored code.

Unfortunately, constraint solvers are not suitable to reason about long execution traces. In our case, we configured our multi-path exploration engine to discard execution paths with a trace longer than a given threshold. This parametrization allows us to keep the analysis as simple as possible and computationally feasible.

**Blocking API calls.** Our system uses a mechanism to bypass blocking API function calls. In some cases, the program gets blocked waiting for user input or certain events in the system. For this reason, when certain APIs such as the `read` or `recv` functions are called, instead of letting the program run, we restore the instruction pointer to the return address in the moment of the call. Also, we fill the output buffers and output values with fake data, and taint those buffers. This approach allows us to successfully run the samples that would otherwise need some network simulation or external interaction.

**String comparison optimization.** The last optimization implemented is related to string comparisons, an operation commonly performed by malware to parse commands (e.g., IRC or HTTP bots). These string comparisons are commonly implemented by means of system API calls such as `strcmp` and `strlen`.

Some functions can return different non-tainted constant values depending on the path followed during execution (that may depend on tainted conditional jumps). Nevertheless, since the code is outside the boundaries of the protected code, these paths will not be explored. For instance, an `strlen`-like function will have a character counter that is incremented for each non-null character found in the string. This counter is a non-tainted value, and our approach does not explore any alternative paths inside the function. As a result, the function will return a non-tainted constant value although the input parameter is tainted.

In order to deal with this limitation and to minimize the processing overhead in such string operations, we hooked 15 different string comparison functions in several DLLs in order to taint the return value of the function whenever a tainted value is provided as input parameter to it.

### 2.3 Heuristic to guide the multipath exploration

One way to reduce the state space and thus the complexity of multipath exploration is to apply heuristics in order to determine which paths should be expanded first. We propose a heuristic based on the intuition that, for a packer protected using the shifting-decode-frames technique, a subset of its execution paths (i.e., one or several instructions in the program) can trigger the execution of a region (e.g., function or memory page). Therefore, in these cases, it is not necessary to explore all the possible paths in order to fully unpack all the content of a binary.

First, our system extracts all the executed code and unpacked memory regions from a single-path execution trace in order to recover as much code as possible. Then, it analyzes this code and determines the instructions that reference locations in the program that have not been unpacked yet. The system then constructs the call graph and control flow graph of the trace and finds the paths that lead to interesting instructions, and finally it provides this information as input to our multi-path exploration engine in order to prioritize the execution of certain paths that would trigger the unpacking of new regions of code. The next sections detail how this process is performed.

**Dumping unpacked memory regions.** Our framework monitors memory writes and execution, and allows us to dump the unpacked and re-packed memory regions after each run. Once we obtain a complete memory dump, we filter it in order to keep only the regions susceptible of being explored in a multi-path fashion. In order to do this, we first indicate which regions we want to explore, and then we generate a filtered memory dump containing (i) the memory blocks that overlap those regions, and (ii) all the execution blocks traced for those regions.

**Disassembly and translation to intermediate language.** In order to analyze the memory dumped by our tool, we implemented our custom disassembly engine to process the unpacked frames of code. This engine is based on the `binutils` disassembly interface and the `libdisasm` library.

First, for each execution block recorded during the analysis of the packer, we perform a linear sweep disassembly. Execution blocks do not contain any instruction that affects the control flow of the program and therefore a linear-sweep algorithm will always successfully extract the code for these blocks. Second, for each conditional jump pointing to blocks that were not executed, we disassemble the target blocks if they are located in memory already dumped. In this case, we follow a recursive-traversal algorithm in order to disassemble as many instructions as possible from the non-executed parts of the unpacked frames, following any jump, conditional jump or call instruction found. Finally, this code is translated to Vine Intermediate Language (Vine IL) for further processing.

**Obtaining interesting points in the code.** Next, we build the Control Flow Graph for every function found in the disassembled code. We then process the result in order to find points in the code that may trigger the unpacking of other regions of code.

– **Control flow instructions.** Control flow instructions (`jmp`,`call`, and `cjmp`) alter the execution flow of the binary and therefore are susceptible of triggering the unpacking of new frames of code. First, if a non-conditional control flow instructions is executed, then the address pointed by the instruction will be executed next. In the case of `cjmp` instructions, it is possible to find cases in which only one of the branches is executed. Nevertheless, considering that we also disassemble non-executed instructions extracted from the unpacked memory frames, we can also find `jump` and `call` instructions that lead to regions of code not previously observed.
– **Direct memory addressing.** Instructions that access a memory address not previously unpacked can trigger the decryption of a new region.
– **Indirect function calls.** Indirect function calls constitute a problem in multi-path exploration. When the register containing the call address is tainted, we need to reason about all the possible values that it can adopt. In our case, we have simplified this problem by concretely evaluating the call address regardless of its taint value. In order to allow our system to explore different targets for the call, we consider these instructions as interesting points in the program. Our engine will try to explore all the different paths that drive the execution to this point, since they may write different values over the register or memory address used in the indirect call.
– **Constants.** Finally, we also analyze the constant values provided as immediate values in the code and check if they may reference a memory address contained in the original code. This approach allows us to consider potential register-indirect or memory-indirect addressing operations.

**Finding interesting paths.** Once we have identified the interesting points in the code, we can distinguish three different cases:

– Non-conditional instructions that were executed and triggered the unpacking of a region of code. Examples are direct memory addressing operations, constants, or unconditional jumps. We discard these points in the code since they are no longer interesting for guiding the execution.
– Conditional jumps in which only one of the possible branches was executed. In these cases, we notify our engine that the alternative branch is an interesting point that should be reached in the next iteration.
– We include any instruction that can potentially trigger the unpacking of a new region, if it is located at a memory address not executed before.

Finally, functions calls represent a special case that must be considered. For instance, there may be a case in which a fully unpacked function (that was executed) has unexplored paths that drive to new regions of code. There will be one or several points in the code that trigger a call to such function. Even if all these points were executed during previous runs, there are still unexplored paths in the function so we need to keep them in the list of interesting points. This can be applied recursively to all the inter-procedural calls we find in the code.

Once we have identified the list of interesting points in the code, we compute the paths that reach each of them. Whenever a loop in the CFG is detected, we consider two possible paths: one that enters the loop, and another one that does not satisfy the loop condition. We keep iterating the ancestor basic blocks until we reach the function entry point. The final result will be a sequence of *(cjmp,address)* pairs. For each conditional jump, we indicate the address that should be executed next in order to reach the interesting point in the code.

Eventually, there might be several different paths reaching interesting points in the code. Instead of simplifying the list, we keep all the possible paths because they might introduce different path restrictions during execution. In fact, many of the paths computed will not be feasible (i.e., there is no possible assignment for the variables in order to force the path). This feasibility will be tested by the constraint solver during multi-path exploration.

The output of our system is a complete list of the interesting points that can be reached for each of the two possible branches of each *cjmp*. This list is provided as input to the multi-path exploration engine to guide the execution to the interesting parts of the code.

**Queries to the SMT solver.** Whenever a tainted conditional jump is executed, we check if it is present in the list of interesting conditional jumps computed in the previous phase. If the *cjmp* is present in the list, we inspect the number of interesting points that can be reached from each of its paths. Then, we query the SMT solver:

- If the two paths drive to interesting points.
- If only one of the paths leads to an interesting region, but it is not the path taken by default.

If the solver cannot provide a feasible solution, we query the solver again ignoring the path restrictions imposed by the execution trace. If there is a feasible set of values that can be forced in order to follow the alternative path, we create a snapshot and decide the next path to execute.

**Path selection algorithm.** In order to select the next path to execute, we iterate the execution tree in Breath First Search order. This approach allows us to incrementally expand all the paths in the execution tree. More specifically, we select the first path that meets the following conditions:

- The path has been forced less times than the rest of paths.
- If several paths have been forced the same number of times, we prioritize those that were solved by the SMT solver in a consistent manner.

This approach allows us to avoid the recursive exploration of loops, in cases in which there are other paths that will reach the same region more efficiently.

During exploration, we update the list of interesting paths whenever a new memory region is unpacked, removing all the entries that refer to the region.

**Path brute-forcing.** In order to avoid exploring repeatedly the same paths in cases in which there is a complex logic with loops, we limit the maximum

number of times that a path can be forced. When we reach this limit, we query the list of conditional jumps we obtained from static analysis, and try to force the execution of conditional jumps that have never been tainted. Since the branch is not tainted, the SMT solver cannot be queried to compute a set of values to force the branch consistently. While this method to force the execution may result into an undetermined behavior or the instability of the process, there are cases in which this unsound approach lets the system reach other interesting regions of code. For instance, a command parsing routine may divide the input strings into tokens and have a complex parsing logic with plenty of loops. There may be cases in which a loop has to be repeated many times (i.e. loop condition is not tainted). If this loop includes tainted branches and a complex logic inside it, it would unnecessarily make the system expand the execution tree too many times. In these cases, when we reach a certain limit of expansions for each conditional jump, our approach forces the exit of the loop and continues execution. A similar case occurs when a loop variable is not tainted itself, but it is set to a constant value (that triggers the exit) when a specific path is followed. This path may only be triggered once we fully explore inner loops, growing the execution tree excessively. In this case our system will inconsistently force the path to reach this point before expanding further the tree. A different case may occur when the variable is updated using instructions that do not involve tainted values (e.g., `inc`, or `add` an immediate value). In this last case, our approach would force the exit of the loop even if the variable is never set with the correct value.

In conclusion, if a certain memory region can be reached from different execution paths, even if the constraint solver is not capable of providing a feasible set of values, our approach will reach the region if there is at least one path that can be forced in a consistent or inconsistent manner, always trying to maintain system consistency to avoid exceptions and system instability.

Also, in cases like page-granularity protection, we only need to trigger a subset of the paths in order to reach all the code pages, avoiding to explore the rest of paths and thus reducing the complexity of the problem.

## 3 Evaluation

In order to evaluate our approach, we implemented our engine on top of TEMU, totalling 7,500 C/C++, 1,300 Python and 500 OCaml lines of code.

In this section we present three different case studies corresponding to packers that protect samples at different granularity levels. On the one hand, *Backpack* is a packer proposed by Bilge et al. [22] that protects the binary with function-level granularity. On the other hand, Armadillo is a well-known commercial packer that allows to protect binaries with a page granularity.

### 3.1 Backpack

In order to test our approach against *Backpack*, we downloaded the source code of the Kaiten IRC bot, reported to be distributed using the *shellshock* bash vulnerability[4]. This sample connects to an IRC channel and receives commands

[4] `http://blog.trendmicro.com/trendlabs-security-intelligence/shellshock-vulnerability-downloads-kaiten-source-code/`. (Accessed: 2015-11-13)

| | Iteration 0 | Iteration 1 | Iteration 2 | No heuristics |
|---|---|---|---|---|
| Functions unpacked | 5/31 | 11/31 | 27/31 | 8/31 |
| Interesting points | - | 52 | 96 | - |
| Cjmps | - | 36 | 110 | - |
| Snapshots | - | 167 | 544 | 6015 |
| Tainted-consistent cjmps | - | 161 | 525 | 5888 |
| Tainted-inconsistent cjmps | - | 6 | 19 | 127 |
| Untainted cjmps | - | 0 | 40 | - |
| Long traces discarded | - | 6 | 0 | - |
| Time | 5m | 24m | 1.2h | 8h |

**Table 1.** Results obtained for the Kaiten malware packed with backpack.

to perform actions such as remote command execution or network flooding. Backpack is designed to protect the binary at compile time and it is implemented as an LLVM plugin to protect C programs. However, due to a limitation of the plugin, to successfully compile Kaiten using Backpack we had to modify the command dispatching routines of the malware to substitute function pointers with direct calls. Given the functionality of the malware, we configured our system to taint network input considering the `recv`, `connect`, `read`, `write` and `inetaddr` system API functions. Also, we parametrized our system to expand each tainted conditional jump a maximum of 8 times. Once this limit is reached, our system inconsistently forces the conditional jumps that were visited but not tainted.

Table 1 shows the results obtained for this experiment. The sample consists of 31 protected functions that implement a total of 22 different commands, triggered by IRC commands and private messages. The unpacking is performed iteratively. In the first iteration we run the malware without applying any multi-path exploration, revealing only 5 out of 31 functions.

Our heuristic engine reported 52 interesting points and 36 conditional jumps in the code. In the first multi-path iteration, 6 new functions were unpacked requiring a total of 167 snapshots. These functions correspond to the 6 different IRC commands implemented by the bot. One of these commands is `PRIVMSG`, that triggers the execution of a function that processes the rest of arguments to trigger different bot commands. Once this function was unpacked in the first multi-path iteration, our static analysis found 96 interesing points in the code and 110 conditional jumps that could drive the execution to functions not yet unpacked. In the last iteration, 27 functions were triggered requiring 525 snapshots. These results show that a concrete execution only reveals a little portion of the real contents of the binary. Also, the heuristic allows to discover new functions in the binary exploring a relatively low number of paths.

Table 1 also shows the number of tainted conditional jumps forced consistently and inconsistently. The number of inconsistently forced cjmps is very low in both cases. Our local-consistency based exploration algorithm and the rest of domain-specific optimizations allow us to tolerate certain inconsistencies with the rest of the system, improving the ability of the approach to force locally

consistent paths. Nevertheless, there are still a few cases in which inconsistent assumptions allow to explore alternative paths that otherwise would be infeasible to explore.

We can also observe that our system recovered the code of almost all the protected functions. More specifically, the 22 main commands were revealed, and only 4 helper functions remained protected due to the early termination of the process. In the last multi-path exploration run, up to 40 untainted conditional jumps were forced inconsistently in order to trigger the unpacking of new functions. These cases correspond to non-tainted conditional jumps that were identified by our heuristic engine as points that could potentially lead to the unpacking of still protected regions of code. These inconsistencies caused the process to terminate when trying to access inexistent strings.

The last row shows the total time required in order to run the python and OCaml code in charge of postprocessing the execution traces, computing the heuristic, and the multi-path exploration itself. For this sample, the scripts related to the heuristic accounted for the 18% of the total processing time.

The last column shows the results when only the domain specific optimizations were applied (no heuristics were used for path selection). In this case, we let the system run for a total of 8 hours. In this time, the system explored up to 6,000 conditional branches, but was only able to recover 8 functions.

### 3.2 Armadillo

Armadillo is one of the most popular packers among malware writers. It allows to protect binaries with page granularity. This technique, also named *CopyMem-II*, consists of creating two separate processes. The first process attaches to the second one as a debugger, capturing its exceptions. When this process starts the execution at a region not present in memory, an exception is produced and the debugger process takes control. This process makes sure that the exception corresponds to a protected memory page, and then it decrypts the page and on the memory of the debugged process, protecting again the previously executed memory page so that it cannot be collected by an analyst. Following this scheme, only one single page of memory is present in memory at any given time, making extremely difficult for an analyst to recover the entire code of the malware.

We used Armadillo 8.0 to protect two different samples with several pages of code and a complex internal logic. These samples belong to the SDBot and the SpyBot malware families. These families of bots typically connect to IRC servers and accept complex IRC commands. However, only the code of the requested functionality is decrypted in memory. Moreover, these specific samples present a very complex command parsing routine that triggers, at different points, code in memory pages that cannot be reached in any other way. We selected these samples in order to properly test our heuristics, and to demonstrate how our optimizations allow to reduce the complexity of multi-path exploration allowing to drive the execution towards the most interesting points in the execution tree, recovering all the code pages efficiently.

In order to measure the complexity of these samples, we applied the IDA-Metrics[5] plugin. The most complex function in SDBot has 417 branches and a cyclomatic complexity of 321. Overall, it has 104 functions with a total cyclomatic complexity of 674. SpyBot, in contrast, has its command parsing routine spread in 4 functions, and although the most complex function presents a cyclomatic complexity of 135, its overall complexity is 953, significantly higher than SDBot.

Table 2 shows the results obtained for the SDBot sample protected by Armadillo. The malware contains 7 pages of code, but a first concrete execution only reveals code in the first and last pages, leaving a total of 5 protected pages. To fully recover every page, we needed to run our engine in 3 iterations. Also, for this sample, it is strictly necessary to trigger some specific paths inside the command parsing routine in order to reach certain pages of code. This function was reached in the first multi-path run, that revealed 2 more memory pages. A second run revealed 2 more pages that were reached through the function, and the last run reached the last memory page. We can observe that the number of interesting points (i.e. targets in the control flow graph that trigger the unpacking of a previously unseen region) is always very low because only a few paths linked the code in one page to code in the next. Although this means that it is only possible to reach these pages by executing those points in the code, it also means that our system only needs to focus on steering the execution towards those points in the code, ignoring all other paths that are not related to them. This brings a very large improvement over a classic multi-path execution approach.

Despite the high number of conditional jumps reported in Table 2, we can observe that the number of snapshots remains low because our heuristics and optimizations allow to priorities the paths and to limit the depth of the execution tree in presence of loops. The number of inconsistent queries is lower than the number of consistent queries, as a result of the local consistency model described that allows tainted variables to adopt free symbolic values (i.e., not tied to global restrictions). We can also observe that our system only needed to force one untainted conditional jump in the second and third iterations, in order to force the exit of complex loops in the command parsing routine.

The last column shows the results for multi-path exploration without heuristics. Similarly to the previous experiment, we let the system run for 8 hours and observed that although the number of expanded conditional jumps was much higher (3660 snapshots), only 4 pages were recovered.

Finally, Table 3 shows the results obtained for the SpyBot malware. In this case, the command parsing routine is spread in several functions that combined together present a more complex logic than SDBot and an higher number of untainted conditional jumps. This sample was unpacked in 2 multi-path exploration runs. In this case, the concrete execution revealed 3/9 code pages, the first multi-path exploration revealed 8 pages, and finally one last multi-path exploration reached the last page.

[5] `https://github.com/MShudrak/IDAmetrics`

| | Iter. 0 | Iter. 1 | Iter. 2 | Iter. 3 | No heuristics |
|---|---|---|---|---|---|
| Pages unpacked | 2/7 | 4/7 | 6/7 | 7/7 | 4/7 |
| Interesting points | - | 3 | 2 | 7 | - |
| Cjmps | - | 65 | 162 | 264 | - |
| Snapshots | - | 14 | 366 | 367 | 3974 |
| Tainted-consistent cjmps | - | 13 | 295 | 296 | 3660 |
| Tainted-inconsistent cjmps | - | 1 | 71 | 71 | 314 |
| Untainted cjmps | - | 0 | 1 | 1 | - |
| Long traces discarded | - | 1 | 14 | 14 | - |
| Time | 30m | 2.2h | 2.8h | 3.2h | 8h |

**Table 2.** Results obtained for the SDBot malware and Armadillo 8.0.

| | Iteration 0 | Iteration 1 | Iteration 2 | No heuristics |
|---|---|---|---|---|
| Pages unpacked | 3/9 | 8/9 | 9/9 | 6/9 |
| Interesting points | - | 26 | 1 | - |
| Cjmps | - | 163 | 214 | - |
| Snapshots | - | 113 | 153 | 4466 |
| Tainted-consistent cjmps | - | 17 | 31 | 4096 |
| Tainted-inconsistent cjmps | - | 96 | 122 | 370 |
| Untainted cjmps | - | 17 | 34 | - |
| Long traces discarded | - | 9 | 34 | - |
| Time | 30m | 3h | 2.75h | 8h |

**Table 3.** Results obtained for the SpyBot malware and Armadillo 8.0.

We can observe that the number of conditional jumps that can drive the execution to the interesting points is similar but there is a higher number of interesting points. Nevertheless, for this sample there was one single transition point to reach the last code page, located deep into the last command parsing routine of the bot. In this experiment, we can notice that the number of queries that the SMT solver was not able to solve is higher, resulting into more tainted and untainted conditional jumps forced inconsistently. Again, like in previous cases, when the system was run without heuristics, only 6 pages were recovered after 8 hours, requiring a much higher number of snapshots. In this last case, the postprocessing scripts represented the 59% of the processing time.

## 4 Discussion

In order to evaluate our approach we have presented three case studies corresponding to samples with complex routines, hundreds of conditional jumps depending on program input, and many string parsing loops. The results of our experiments show that, by adding several domain specific optimizations and heuristics, it is feasible to apply multi-path exploration to unpack complex binaries protected with shifting-decode-frames.

We selected three case studies in order to test our approach, using two different packers for protection. Although the number of tests is low, we selected

representative samples with complex logic and different protection granularities. In fact, most of the packers reveal all the protected code at once and only few present this advanced protection mechanism. Beria applies the same approach as Armadillo, but presents a lower overall complexity. Unfortunately, it is not a common packer, and thus we found no interesting samples available.

We only tested one sample protected with Backpack because was developed for GNU/Linux and it requires the source code of the malware in order to apply the protection at compilation time. Given this restriction, we selected the most complex GNU/Linux malware source code we could compile with Backpack.

In the case of Armadillo, we needed to meet several requirements in order to properly test our approach and heuristics. First, we needed samples with complex routines depending on program input. These samples had to trigger the execution of new regions of code (not executed in a single concrete run), only after executing a fairly complex amount of code. Also, we selected samples that did not already present a custom packing routine. Otherwise, only that routine would be protected by Armadillo, greatly simplifying our job and not providing a challenge for our system. Similarly, we had to discard samples that decode and inject all their code into another process once the execution starts, as well as droppers, downloaders, and simple spyware due to their simplicity.

Our approach is based on whole system emulation, which has a number of well-known limitations. For instance, red-pills can be used to determine if the execution environment in which it runs is a virtual/emulated environment or a real machine. In fact, Paleari et al. [23] proposed a method to automatically discover and generate red-pills in system emulators. In particular, during this project we found two implementation errors in the Dynamic Binary Translation engine of QEMU that affected all its versions and impeded the correct emulation of the Armadillo packer. In this context, several publications and projects [3, 24, 25] have reported the incapacity of emulators to correctly execute the Armadillo packer. We solved this issue and reported it to the QEMU developers.

Finally, although the samples evaluated in this study were not affected by the following techniques, complex packers may leverage them to hinder our approach.

- **Calling convention violation.** Malware can violate calling conventions in order to obfuscate the code. If these techniques are employed to obfuscate API function calls (e.g., stolen bytes), our tracing mechanisms could fail to locate string parsing functions, affecting some of our optimizations.
- **Alternative methods to redirect control-flow.** In order to evade multi-path exploration, malware samples may potentially use alternative methods to redirect the control flow: alternative combinations of instructions such as `push` + `ret`, indirect calls, `call` + `pop` + `push` + `jmp`, SEH or VEH based redirection, opaque predicates in branch instructions, or even obfuscating the computation of triggers [26].
- **Resource exhaustion**. Our techniques reduce the computing overhead of multi-path exploration. Nevertheless, creating memory snapshots and querying SMT solvers over long traces still requires significant computing resources. A packer may increase the complexity of the code affecting impact-

ing the performance of multi-path exploration. A malware writer may design a complex CFG with a high number of loops and conditional jumps specifically crafted to increase the number of paths to explore with our heuristic.

- **Nanomites.** This technique consists in replacing conditional branch instructions by software interrupts (e.g. `INT 3`) that cause the execution to break. A parent process intercepts the exception and then overwrites the conditional jump. A more complicated example involves redirecting the execution of the child by evaluating its context (state of the `EFLAGS` register) and redirecting its execution to the appropriate address, without even replacing the interrupt instruction with the original instruction. This technique would break taint propagation and prevent us from successfully reconstructing the CFG.

## 5 Related Work

Manual unpacking requires a substantial reverse engineering effort. Consequently, many researchers have focused on generic unpacking in recent years. Both dynamic and static [7] approaches have been proposed, but due to the complexity of static approaches, most of the authors have focused on dynamic analysis, installing drivers in the system [6, 27] or tracing the execution [3].

Some of these systems rely on heuristics or monitor coarse-grained events [27], while others monitor memory writes and memory execution at different granularity levels [3,6,28,29], compare the static and run-time version of the memory [2], perform statistical analysis [5], or measure the entropy variation [4].

Other approaches rely on hybrid static and dynamic analysis [30]. Virtualization based packers constitute a special category of protection techniques. Several authors have focused on unpacking these packers from different perspectives [31–33]. Nevertheless, these protection engines are a different challenge that require other techniques in order to recover the original code.

Transparent execution [1, 34, 35] is focused on dealing with malware capable of detecting the analysis environment and modifying its execution to evade detection. Nevertheless, these techniques do not explore the different execution paths that a binary may have. Bilge et al. [22] demonstrated that this limitation can be leveraged by an attacker in order to defeat unpackers that assume that all the code will be present in memory at some moment in time.

In order to improve the test coverage in malware analysis, Moser et al. [14] proposed a system to explore different execution paths based on taint analysis and symbolic execution. Our work is built on top of this research, adding a set of optimizations and heuristics to deal with a specific use-case.

Almost in parallel, Song et al. [15] developed a platform for binary analysis. This platform was used in many different follow-up works, including identification of trigger-based behaviour [10], reasoning about code paths in malware using mixed concrete and symbolic execution [11], or even triggering the unpacking routine of environment sensitive malware [12]. Another closely related project is $S^2E$ [18], a platform that introduces the concept of selective symbolic execution (application of symbolic execution to only certain memory regions)

and execution consistency models. Schwartz et al. [16] summarized the challenges and limitations that affect efficiency and feasibility of symbolic execution. Taint policies and the sanitization of tainted values have a direct impact on over-tainting and under-tainting errors. Indirect memory accesses with symbolic addresses, jump tables, or the size of the constraint systems are aspects that have no clear solution. Finally, X-Force [13] is a system capable of forcing execution paths inconsistently and recovering from execution errors by dynamically allocating memory and updating related pointers. More specifically, they focus on 3 different goals: (i) constructing the control flow graph of a binary, type reverse engineering, and discovering hidden behavior in malware. Our approaches share some concepts, such as forcing the execution inconsistently. However, their main contribution is a technique to recover from errors (which is not as important in our domain), while our contributions are a set of domain-specific optimizations, and a heuristic to drive the exploration. Also, we focus on applying multi-path exploration to unpacking samples with a complex command parsing logic, a problem that typically presents a high complexity. To this aim, our approach mixes consistent and inconsistent multi-path exploration to maximise system consistency in order to reach deep execution paths. Overall, our goal is not to improve multi-path exploration, but to show if and how this technique can be used for unpacking, and which customizations are required in order to improve its results. To sum up, all these approaches suffer from the well-known path explosion problem [21]. This limitation makes necessary to develop heuristics and optimizations in order to improve the feasibility of multi-path exploration, and this is the main contribution of our paper.

## 6 Conclusions

In previous sections we have described the domain-specific optimizations and heuristics that can be implemented over multi-path exploration to unpack shifting-decode-frames protectors. We have evaluated our approach over three different case studies covering Backpack, a function granularity based packer, and Armadillo, a well-known packer that protects binaries with a page-granularity. Our test cases cover different samples with complex command parsing logic.

Multi-path exploration has been addressed by several researchers but it is not generally used for real-scale malware analysis due to its technical complexity and its limitations. Our results show that it is possible to apply optimizations and heuristics to multi-path exploration in order to address specific problems such as the malware protection technique covered by this study.

## Acknowledgements

We would like to thank the reviewers for their insightful comments and our shepherd Brendan Dolan-Gavitt for his assistance to improve the quality of this paper. This research was partially supported by the Basque Government under a pre-doctoral grant given to Xabier Ugarte-Pedrero.

## References


1. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on Computer and communications security, ACM (2008) 51–62
2. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: Proceedings of the $22^{nd}$ Annual Computer Security Applications Conference. (2006) 289–300
3. Kang, M., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proceedings of the 2007 ACM workshop on Recurring malcode. (2007) 46–53
4. Cesare, S., Xiang, Y.: Classification of malware using structured control flow. In: Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107, Australian Computer Society, Inc. (2010) 61–70
5. Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., Lee, W.: Eureka: A Framework for Enabling Static Malware Analysis. In: Proceedings of the European Symposium on Research in Computer Security (ESORICS). (2008) 481–500
6. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, IEEE (2007) 431–441
7. Coogan, K., Debray, S., Kaochar, T., Townsend, G.: Automatic static unpacking of malware binaries. In: 16th Working Conference on Reverse Engineering, 2009., IEEE (2009) 167–176
8. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: [SoK] Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society (May 2015)
9. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Proceedings of the $23^{rd}$ Annual Computer Security Applications Conference (ACSAC). (2007) 421–430
10. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Botnet Detection. Springer (2008) 65–88
11. Brumley, D., Hartwig, C., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin, H.: Bitscope: Automatically dissecting malicious binaries. School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-07-133 (2007)
12. Jia, C., Wang, Z., Lu, K., Liu, X., Liu, X.: Directed hidden-code extractor for environment-sensitive malwares. Physics Procedia **24** (2012) 1621–1627
13. Peng, F., Deng, Z., Zhang, X., Xu, D., Lin, Z., Su, Z.: X-force: Force-executing binary programs for security applications. In: Proceedings of the 2014 USENIX Security Symposium, San Diego, CA. (2014)
14. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: IEEE Symposium on Security and Privacy, 2007, IEEE (2007) 231–245
15. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: Information systems security. Springer (2008) 1–25
16. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: IEEE Symposium on Security and Privacy 2010, IEEE (2010) 317–331

17. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the $8^{th}$ USENIX conference on Operating Systems Design and Implementation (OSDI). Volume 8. (2008) 209–224
18. Chipounov, V., Kuznetsov, V., Candea, G.: S2e: A platform for in-vivo multi-path analysis of software systems. ACM SIGARCH Computer Architecture News **39**(1) (2011) 265–278
19. Brumley, D., Wang, H., Jha, S., Song, D.: Creating vulnerability signatures using weakest preconditions. In: $20^{th}$ IEEE Computer Security Foundations Symposium (CSF), IEEE (2007) 311–325
20. Leino, K.R.M.: Efficient weakest preconditions. Information Processing Letters **93**(6) (2005) 281–288
21. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Tools for Practical Software Verification. Springer 1–30
22. Bilge, L., Lanzi, A., Balzarotti, D.: Thwarting real-time dynamic unpacking. In: Proceedings of the $4^{nd}$ European Workshop on System Security, ACM (2011) 5
23. Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D.: A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In: Proceedings of the USENIX Workshop on Offensive Technologies (WOOT) Vol 41. (2009) 86
24. Deng, Z., Zhang, X., Xu, D.: Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In: Proceedings of the 29th Annual Computer Security Applications Conference, ACM (2013) 289–298
25. Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., Vigna, G.: Efficient detection of split personalities in malware. In: Network and Distributed System Security Symposium (NDSS). (2010)
26. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: Network and Distributed System Security Symposium (NDSS). (2008)
27. Guo, F., Ferrie, P., Chiueh, T.C.: A study of the packer problem and its solutions. In: Proceedings of the 2008 Conference on Recent Advances in Intrusion Detection (RAID). (2008) 98–115
28. Stewart, J.: Ollybone: Semi-automatic unpacking on ia-32. In: Proceedings of the $14^{th}$ DEF CON Hacking Conference. (2006)
29. Kim, H.C., Inoue, D., Eto, M., Takagi, Y., Nakao, K.: Toward generic unpacking techniques for malware analysis with quantification of code revelation. In: The $4^{th}$ Joint Workshop on Information Security. (2009)
30. Caballero, J., Johnson, N., McCamant, S., Song, D.: Binary code extraction and interface identification for security applications. In: Proceedings of the 17th Annual Network and Distributed System Security Symposium, ISOC (2009) 391–408
31. Rolles, R.: Unpacking virtualization obfuscators. In: 3rd USENIX Workshop on Offensive Technologies.(WOOT). (2009)
32. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: $30^{th}$ IEEE Symposium on Security and Privacy, IEEE (2009) 94–109
33. Coogan, K., Lu, G., Debray, S.: Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In: Proceedings of the 18th ACM conference on Computer and communications security, ACM (2011) 275–284
34. Vasudevan, A., Yerraballi, R.: Cobra: Fine-grained malware analysis using stealth localized-executions. In: IEEE Symposium on Security and Privacy. (2006) 15–pp
35. Kang, M.G., Yin, H., Hanna, S., McCamant, S., Song, D.: Emulating emulation-resistant malware. In: Proceedings of the $1^{st}$ ACM workshop on Virtual machine security, ACM (2009) 11–22