Maitland: Lighter-Weight VM Introspection to Support Cyber-Security in the Cloud

Chris Benninger*, Stephen W. Neville†,
Yağız Onat Yazır*, Chris Matthews*, Yvonne Coady*
*Department of Computer Science
†Department of Electrical and Computer Engineering
University of Victoria, Victoria, B.C., Canada

***Abstract*—Despite defensive advances, malicious software (*malware*) remains an ever present cyber-security threat. Cloud environments are far from malware immune, in that: i) they innately support the execution of remotely supplied code, and ii) escaping their virtual machine (VM) confines has proven relatively easy to achieve in practice. The growing interest in clouds by industries and governments is also creating a core need to be able to formally address cloud security and privacy issues. VM introspection provides one of the core cyber-security tools for analyzing the run-time behaviors of code. Traditionally, introspection approaches have required close integration with the underlying hypervisors and substantial re-engineering when OS updates and patches are applied. Such heavy-weight introspection techniques, therefore, are too invasive to fit well within modern commercial clouds. Instead, lighter-weight introspection techniques are required that provide the same levels of within-VM observability but without the tight hypervisor and OS patch-level integration. This work introduces Maitland as a prototype proof-of-concept implementation a lighter-weight introspection tool, which exploits paravirtualization to meet these end-goals. The work assesses Maitland's performance, highlights its use to perform packer-independent malware detection, and assesses whether, with further optimizations, Maitland could provide a viable approach for introspection in commercial clouds.**

***Keywords*-clouds; malware; VM introspection**

I. Introduction

Cloud computing represents the most recent stage in the software industry's continuous evolution towards more efficient, resilient, and feature-filled software systems. The establishment of commercial clouds is fueling a strong paradigm shift with respect to how software systems are deployed as a result of their innate cost-effectiveness, opportunities for increased system reliability and availability, etc. Malicious software (or *malware*) has, of course, evolved in parallel.

More particularly, as cyber-defenses have improved, malware writers have needed to become increasingly adept at hiding (or obfuscating) their malcode such that it retains effectiveness despite the current generation of cyber-defenses. The rise of cyber-crime [1], in part fueled by the ease with which cyber-attacks can now be converted into monetary value, has fostered the maturing of underground malware economies in which professional-level malware development kits (MDK) can be easily found and purchased. These, in turn, are fueling an overall year-over-year increase in the observed complexity and richness of in-the-wild malcode, as facilitated by these MDK's easy to use GUI-driven interfaces [2].

Cloud environments offer particularly attractive malware targets as they incorporate vast numbers of computing resources, high network bandwidths, and are increasingly becoming the operational home to many high-valued software systems and services. Attacks targeting clouds, therefore, offer significant opportunities to: gain control over resources, extract proprietary and private information, engage in targeted denial-of-service attacks, etc. Moreover, the innate structure of clouds allows for the uploading and execution of remotely supplied code, and implicitly relies on VM isolation (or sandboxing) to provide core aspects of security, whereas this is known to be defeatable [3][4][5].

Within the cyber-security community, *static* and *dynamic* analysis have provided the traditional means to address malware, where the former relies on analyzing code prior to its execution and the latter analyzes run-time behavioral features [6]. The ease with which static analysis can be beatable through malcode packing or encryption has driven the cyber-security community to focus more strongly on dynamic analysis methods. In general, dynamic analysis has been supported through VM introspection techniques [7] [8], which provide easy observability into the details of how a given piece of code touches memory, uses OS services, etc. In response to the anti-malware industry's use of VM introspection, malware writers adapted their malcode to "play nice" when it detects it is running within a VM. The wide-scale use of VMs within clouds requires that malware targeting clouds must continue to act maliciously even when detects it is executing within a VM. This in turn, makes applying VM introspection techniques to clouds a potentially attractive cyber-security approach.

A core pragmatic problem arises with current cyber-security VM introspection techniques as they are quite heavy-weight. This in turn, requires direct hypervisor modifications as well as significant re-engineering to track the changes to OS memory structure locations that occur with OS-level updates and patches [9]. However, within commercial clouds applying such heavy-weight introspection is simply untenable given that commercial cloud providers would prefer not to have new code introduced to their operational hypervisors—given the risks entailed for overall cloud stability and reliability. Moreover, commercial providers would likely find it undesirable to

support the costly re-engineering efforts required to provide tailored introspection for the full set of OSes and OS patch-levels that any of their customers may need (or desire) to run.

This does not negate a need within clouds to support VM introspection as a means to foster improved cyber-security. A strong argument can be made for the development of lighter-weight introspection tools that achieve the same end-goals as the prior heavy-weight methods but without their high integration and re-engineering costs. This work introduces a such a lighter-weight introspection tool prototype called Maitland. More particularly, Maitland exploits the on-going shift towards paravirtualization to enable the desired light-weight introspection capabilities. The efficacy of Maitland is shown by demonstrating its use to detect packed code in a manner that is packer-independent. Experimental results show that, with further optimization, Maitland can provide a pragmatic and viable mechanism to address VM introspection needs within commercial cloud environments.

The remainder of this paper is organized as follows. Section II presents the related work specific to the detection of malware. Section III defines the specific end-goals of this work with respect to lighter-weight cloud-based VM introspection. Section IV introduces Maitland and provides details on its light-weight design. Section V presents a pragmatic example in which Maitland is used to detect multi-round packed code as well as Maitland's resulting performance overheads. Section VI discusses Maitland's current limitations. Finally, Section VI-A concludes this work and discusses avenues for future work.

II. Related Work

Static and *dynamic analysis* constitute the two main approaches for detecting in-the-wild malware. Static malware analysis differs from dynamic analysis in the sense that it denotes the searching for malicious intent among sets of binary instructions without executing those instructions [10]. Whereas, dynamic analysis seeks to detect malicious intent by observing what occurs once the instructions are executed [10].

Static analysis has long been known to be easily beaten by applying packing and/or encryption schemes to malicious instruction sequences such that they no longer contain any known malicious patterns (or signatures) [11]. In this context, *packing* refers to the re-ordering or re-spacing of the code in manners that, in a formal sense, fall short of true encryption. Of course, packed or encrypted code is innately not executable and, hence, its malicious effects cannot occur until it has been unpacked or decrypted back into actual machine code mnemonics. Multi-round and multi-layer packing or encryption can also be used to defeat defensive attempts to unravel packed or encrypted malware [12]. As each round or layer can be performed by a different packing or encryption scheme, the complexity of the defenders' problem can be arbitrarily increased. Malcode unpacking (or decryption) can also be done just-in-time as each memory page is required, further complicating its detection. Figure 1 provides an illustrative example of multi-round packing.

Recent studies estimate that at least 70% of new malware variants utilize some form of packing [13] [14] [15], with new packers surfacing almost daily [16]. Hence, it is not surprising that malcode packing was seen in both W32.Sality [17], Symantec's top malicious code family of 2010, and W32.Stuxnet [18], considered one of the most expensive and complex pieces of malware ever created [19]. Common malcode packers include: *Ultimate Packer for eXecutables* (*UPX*) [20], *Themida* [21], and *ASPack* [22].

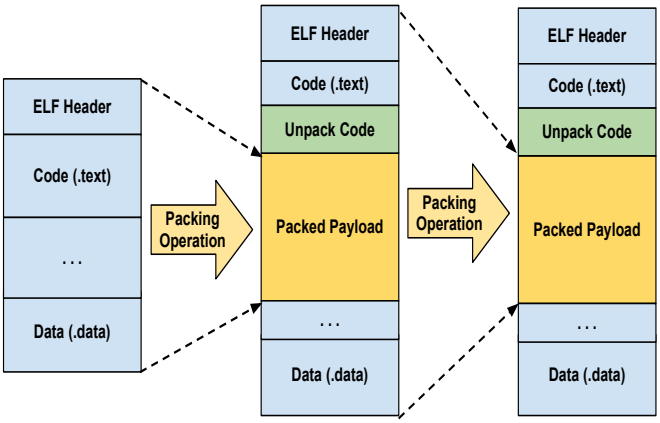


Fig. 1: Example of multiple packing layers.

Even once malcode has been unpacked or decrypted such that it exists as machine level instructions, its true functionality can also be relatively easily hidden through the use of any number of well known code obfuscation techniques. In general, these techniques can focus on changing the structure of the code, e.g. as per polymorphic malware [23]. In addition, they can also change the nature of the code's run-time traces, e.g. as per metamorphic malware [24]. Moreover, simple obfuscations such as the introduction of dummy or null code blocks can be used. Some of these obfuscations can be removed through applying techniques developed for optimizing compilers [25], however, the general problem of arbitrary malcode detection is known to be undecidable [26].

Furthermore, it has become increasingly important to be able to ascertain whether or not given pieces of code may contain time- or logic-bombs, i.e., dormant pieces of malcode functionality that are triggered by predefined inputs or at predefined times. Standard dynamic detection techniques are largely restricted to analyzing information gleaned from executed code paths. Hence, they cannot detect any logic- or time-bombs that may exist in any yet-to-be-run code. VM introspection provides an interesting avenue to combine dynamic and static detection techniques. In this context, Introspection can provide the full set of information available from any executable memory pages as well as the subset of executed instructions. Introspection is also a common method for providing the standard execution information used in dynamic analysis. The core problem is the heavy-weight nature of

existing VM introspection techniques, such as *Patagonix* [27], *Ether* [9], and *Eureka* [14].

III. Application to Cloud Introspection

An innate observation can be made that VMs provide a relatively easy way of detecting when memory pages change from being marked as data pages to being marked as executable pages which is a necessary precursor step before packed or encrypted malcode can perform any malicious actions. Additional cyber-defensive control actions can then be inserted into the hypervisor's execution process to asses potential maliciousness. By definition a newly marked executable page must contain machine-specific mnemonics. This provides a core means to address multi-round or multi-layer malcode packing and encryption techniques in a manner that is packer (or encryption scheme) independent.

The use of virtualization in clouds provides the innate advantage that no user supplied code should ever be executing outside of a VM. Hence, within clouds, introspection can be used to trigger interrupts when data pages are reset to be instruction pages. The process owning the memory pages can then be taken off the OS process schedule until the memory page(s) have been appropriately vetted as being non-malicious. Within clouds, this vetting could easily be performed by a centralized cloud service, thereby fostering improved cloud-wide cyber-security.

Enabling sufficiently light-weight VM introspection capabilities is on the critical path to structuring such a cloud-wide security service within modern commercial cloud environments. Modern paravirtualized VMs offer an attractive avenue to achieve this goal. However, the innate trade-offs between paravirtualized introspection and prior heavier-weight introspection techniques must be assessed. Maitland provides a prototype proof-of-concept implementation to explore these issues. The key end-goals underlying Maitland's construction are:

- To provide full VM introspection capabilities without requiring modified or custom hypervisors,
- To provide independence from the need to re-engineer based on OS-level patches and upgrades,
- To provide a simple and language-independent interface by which a wide variety of cyber-security malware analysis methods can access the introspection capabilities,
- To introduce no new security vulnerabilities,
- To avoid imposing unreasonable overheads into introspected OSes and their executing programs,
- To provide a packer-independent and encryption-independent mechanism of detecting multi-round or multi-layer packed or encrypted malcode.

IV. Maitland

Maitland is a paravirtualization-based tool to facilitate lighter-weight VM introspection. Maitland uses *dirty-page tracing* as a mechanism for detecting encrypted or packed binaries, including those employing multi-round or multi-layer approaches. Our design is intended to be minimally invasive and easily extendable to include arbitrary malware analysis (or other) cyber-security tools. Maitland is designed to be easy to install, deploy, modify, and extend while retaining the capabilities of prior heavier-weight VM introspection solutions.

Maitland uses Xen's paravirtualization extensions to provide a method by which arbitrary, authorized, cyber-security analysis tools can gain access to per-process memory snapshots of running processes within VMs. The timing of these snapshots is under the analysis tools' direction. Although the prototype has been built for Xen, the Maitland approach can be applied to any hypervisor that supports paravirtualization.

Maitland exists as a pair of loadable kernel modules: one for each to-be-introspected guest VM and another for a privileged analysis VM. As a majority of cloud platforms have (or are) moving to run on some form of a paravirtualized Linux kernel [28] [29], Maitland's need for paravirtualization is not viewed as a pragmatic limitation. It should be noted that the requirement of a guest VM component is also not viewed as a limitation in production commercial cloud environments as the cloud customers are likely required to run VMs as supplied and configured by the cloud provider. Therefore, malware that decides to "play nice" when it detects it us running in a VM or Maitland itself only serves to increase the overall security of the cloud. Such levels of control are particular to commercial cloud platforms, and no argument is being made that Maitland serves as a general solution outside of such environments.

A. Architectural Overview

Maitland is a thin layer which spans kernel-space between the privileged VM and any instantiated guest VMs. A cyber-security analysis tool typically exists in user-space on a privileged VM. Such a tool would interface with Maitland through a standard UNIX file-handle using common file-ops and *IOCTL* commands [30]. Figure 2 illustrates Maitland's architecture.

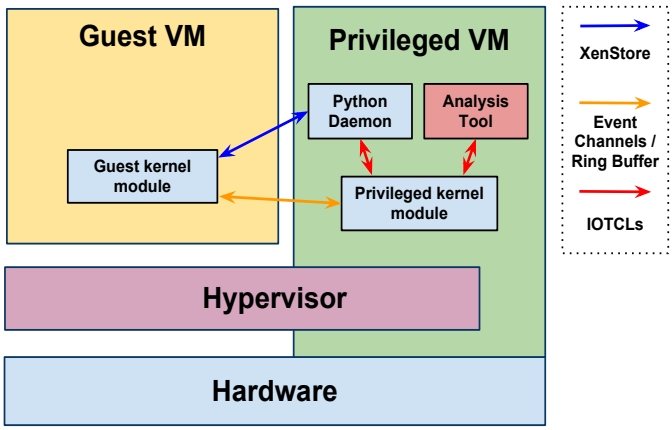


Fig. 2: Maitland's detailed architecture.

Maitland uses *XenStore* [31] and *event channels* as methods for propagating event information between VMs and Xens's *grant table* [32] mechanism to export captured memory snapshots.

B. Detecting Memory Page Unpacking and Decryption

Maitland uses dirty-page tracking as a primary method for detecting executable memory pages that have been produced through unpacking (or decryption) operations. This method is also used by tools such as OmniUnpack [33], Renovo [34] and Justin [12]. This forms the core example of how Maitland can be used to intercept and add control sequences to OS system events that routinely happen inside guest VMs. It also serves as the test case for assessing Maitland's performance impact.

The general idea is that: i) a packed (or encrypted) malicious executable must be unpacked (or decrypted) into machine code mnemonics before it can enact *any* malicious actions, and ii) this innately requires that the relevant memory pages must go from being marked as data pages to being marked as executable pages. By tracking the state of dirty pages associated with a process, Maitland can detect unpacking (or decryption) at the end-point when the malicious machine level instructions come into existence. Tracking of these properties can be achieved in a paravirtualized Linux kernel by: i) intercepting *Memory Management Unit* updates then checking the *page-dirty* flag, and ii) handling page faults then checking the *NX-bit*, as described below.

1) MMU Updates and the Page-Dirty Flag: Modern OSes use page-dirty flags to track which memory pages need to be flushed to permanent storage in order to preserve newly-written data. The page-dirty flag is typically contained in the associated running process' page-table entry. The OS automatically sets this flag as pages are written to. When a program allocates memory or modifies a previous allocation, the MMU detects the operation and notifies the OS via an interrupt (an MMU update). Maitland exploits this mechanism to monitor new allocations, re-allocations and when a page is being set as writable.

Under native Linux environments, MMU updates are handled internally through the actions of the CPU and are signaled via an OS-level interrupt handler. In hardware virtualized environments, the hypervisor typically intercepts the CPU-level interrupt and handles it internally before passing control transparently to the OS within a guest VM. In paravirtualized systems, the guest VM OS is modified to perform a direct hypercall to the hypervisor and request an MMU update. Maitland hooks into this paravirtualized process to ensure that the privileged cyber-security analysis VM is also notified when these events occur. Hence, Maitland exposes which pages are being modified by which programs within which guest VMs to security analysis tools in the privileged VM. If, in the case of a cloud-level security service, it is already known that one such program is untrusted, then Maitland can also be used to set the pages containing the newly written data associated with this process to the unexecutable state, thereby, forestalling their intended malicious actions.

2) Page Faults and the NX Bit: As Maitland can keep track of dirty pages, it must be able to know when a memory page has been masked as executable. The NX bit, also known as the *Non-Execute* bit on Intel x86 [35] architectures and *XD* or *Execute Disable* bit on AMD64 architectures provide hardware-enforced methods for preventing the execution of memory page contents.

When a page marked not-executable contains data being accessed as an instruction, the CPU will issue a page fault. Maitland focuses on such page faults triggered by instruction fetches on not-executable pages. Maitland inserts a callback in the paravirtualization code that interfaces with the hypervisor on such page faults and filters these faults to see if the CR2 register contains a stack pointer of an untrusted process. Page faults making it through these filters are flagged as potential malware unpacking (or decryption) operations.

C. Accessing a Process' Memory Snapshot

Once Maitland has detected an attempted execution of a dirty page, it proceeds to provide a memory snapshot of the offending program to a privileged security analysis VM. Again within commercial clouds, it is possible to ensure that such a VM must exist and must be running on all active cloud servers. Maitland removes the process owning the memory page that caused the observed page fault from the OS scheduler and walks the process' page table to make a full list of all of its existing memory pages. All memory pages mapped to the suspect process' virtual memory space are then mapped into the privileged cyber-security analysis VM's memory space. Any analysis tool running in the privileged VM can then see and assess the suspect process' full memory snapshot. This approach can, of course, be trivially extended to also include all of the typical run-time information used in dynamic malware analysis. Additionally, Maitland can be configured to use other triggers to grab these memory snapshots, the only limitation being that such triggers exist as observable hypervisor events.

D. Responding to a Perceived Threat

In cases whereby the privileged VM cyber-security analysis tools detect a threat or a perceived threat, the hooks that allow Maitland's introspection capabilities can be used to provide a direct control path back into the guest VM's OS. For example, Maitland's control path can be used to terminate at the OS-level a suspect process within a guest VM. Maitland could also be used to, instead, increase the vigilance on a suspected process by passing out run-time information for further analysis. Maitland may also allow the suspect process to continue execution but without committing its results until the process has produced sufficient information to provide the required analysis confidence that it is indeed non-malicious. Additionally, standard checksum approaches could be used to quickly identify well known data such that it avoids any performance penalties.

E. Exploiting Split Drivers

The current Maitland implementation only supports Linux guest VMs, an obvious limitation, as a result of poor Xen paravirtualization support for Windows platforms. Maitland consists of a frontend/backend Xen driver pair. The frontend driver is inserted into the kernel of unprivileged guest VMs and

a backend driver is inserted into the kernel of the privileged cyber-security analysis VM. Maitland's implementation as a set of kernel drivers innately makes it simple to deploy and run, as all that is required is the loading of a set of kernel modules. As portions of Maitland exist within the guest VM kernels, it can observe their internals with high granularity. Traditionally, such within-VM observability has been accomplished via pre-computed *binary offsets* for the relevant internal OS data-structures, i.e., as per *Ether* [9].

However, this creates a core problem as these offsets are known to be intentionally changed by OS vendors at the patch-level. Because Maitland includes a guest VM kernel component, these within-VM OS memory data-structures can be used directly. This much simpler approach does not require knowledge of OS memory structures and, hence, provides independence from OS upgrades and patches. Of course, the guest VM component negates transparency as Maitland's kernel component is potentially detectable by malware. But, works by Quist et al. [36] and Rutkowska et al. [37] have strongly suggested that, in any event, achieving transparency is an elusive goal. Additionally, commercial clouds offer the strong advantage that they can check whether the Maitland kernel component is present and active in any guest VM prior to allowing it to run. This would thereby, innately force malware to be active in the presence of Maitland irrespective of whether or not Maitland's existence is known to the malware.

V. Performance Evaluation

The following subsections describe the performance evaluations undertaken for Maitland. The core intent of these evaluations was to assess whether Maitland, via its paravirtualized introspection approaches, was in the ball park of techniques that could be pragmatically deployed within commercial clouds once further optimized. Hence, we are expressly not claiming the the current prototype implementation is appropriate for real-world deployments today. Instead, the obtained performance numbers suggest that Maitland likely represents a viable solution path worthy of further exploration and optimization.

A. Hardware and Software Environment

All of the performance tests were conducted on a Intel Xeon dual-core 3.4GHz system with 2.5GB of memory. The virtual machine environment used was Xen 4.0.1 and the 2.6.32.27 64-bit Linux kernel with paravirtualization extensions. Test VMs were each allocated 1 virtual CPU and 128 MB of ram.

B. Evaluated Packers

As Maitland currently only supports Linux guest VMs, the selection of compatible packers was limited to: i) UPX [20], one of the most common packers used for malware obfuscation and ii) *gzexe* which comes pre-installed on some Linux distributions. As the detection approach is not dependent on how the packer actually operates, the performance results are easily generalizable to other packing schemes (known or unknown) as well as encryption schemes.

C. Packed Test Programs

For sample packed programs to be detected, two CPU-intensive applications were chosen as well as one memory-intensive one. This was done as memory page fault patterns and frequencies can change significantly depending on the nature of the software applications, with the selected applications chosen to explore these extremes. More specifically, the specific applications were:

- Pi_css5[38] a tool for calculating π, which for evaluation purposes was used to compute π to 2^{17} digits.
- *Gzip*[39] a standard file compression tool, which for evaluation purposes was used to to compress a randomly generated 10MB binary file.
- A custom tool that allocates large memory segments and writes arbitrary random data into them.

D. Security Analysis Tool

Maitland itself does not include a cyber-security analysis component, as it instead provides an interface for such tools. Hence, for the performance test a simple analysis tools was built that employed a Python wrapper script to use grep to search for known binary strings in the memory snapshots Maitland provided. The binary strings to be detected were extracted from our executables prior to their packing. The end-goal of the performance testing was to measure the Maitland induced performance overhead from the start of the unpacking operations until the a priori known binary string was detected in the guest VM's memory pages by the simple security analysis tool. The tool used Maitland to trigger a "kill process" response once the binary string detection occurred.

E. Performance Results

1) Experiment 1: Unpacking Detection: The goal of this first experiment was to determine the most basic issue of how effective Maitland was at detecting known signatures in packed executables. To begin, unique binary strings (or signatures) were extracted from unpacked versions of gzip and pi_css5. The gzip and pi_css5 executable code was then packed with gzexe and UPX, resulting in the four binary files: gzip_upx, gzip_gzexe, pi_css5_upx , pi_css5_gzexe. It was then verified that each packed binary did not include the binary detection signature. This procedure well mimics the process used by malware to hide known signatures through packing and, hence, represents a reasonable approximation of the problem of applying Maitland to detect packed (or encrypted) malware.

We ran each executable both with and without Maitland enabled, while running the simple analysis process. The results in Table I clearly show that the Maitland-enabled analysis process was able to detect, through the page fault tracking process, the instant when the unpacking caused the prior known strings to come into existence within a guest VM memory pages.

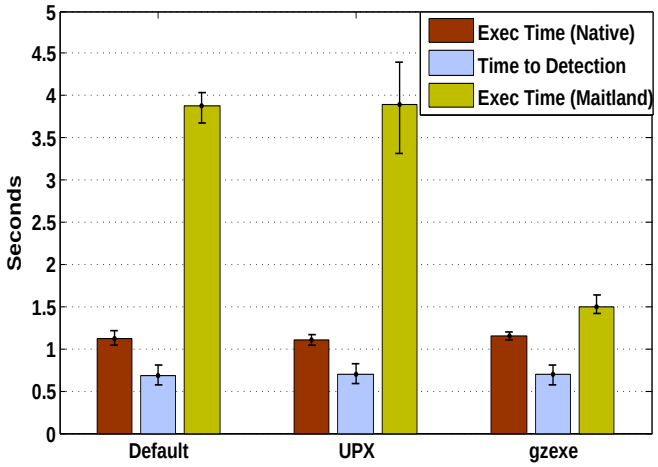
Executable Name	Grep	Maitland+Grep
gzip	detected	detected
gzip_upx	not detected	detected
gzip_gzexe	not detected	detected
pi_css5	detected	detected
pi_css5_upx	not detected	detected
pi_css5_gzexe	not detected	detected

TABLE I: Confirmation of packed executable known signature detection through Maitland enabled page fault tracking.

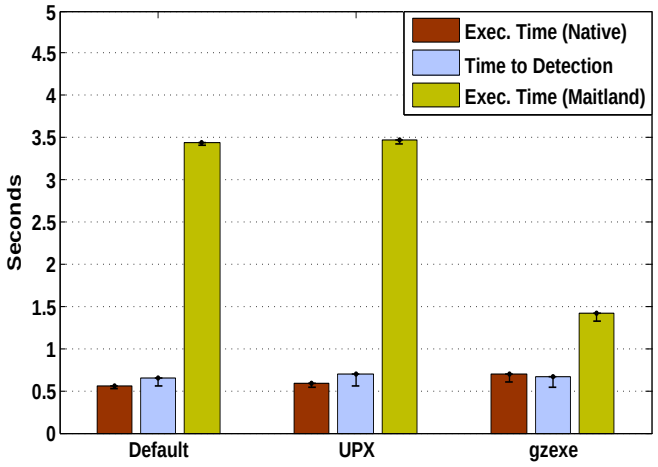
2) Experiment 2: Assessing Overhead Cost: In this experiment, the goal was to quantify the induced overheads of the Maitland prototype. Each of the above executables was first run without Maitland to obtain baseline execution times and then re-run with Maitland enabled with and without the security analysis process enabled. Figure 3 shows the average wall time time for a set of 10 such experiment runs as: native run-time (red), run-time with Maitland and security analysis (blue), and with Maitland but without security analysis (green), with the error bars set to denote the one standard deviation mark for each set of 10 runs. The difference between the blue and green bars in Figure 3 is that blue denotes the executable being terminated once its known signature becomes observable during the unpacking process, while green denotes allowing the process to continue to run with Maitland active through its full execution cycle.

It can be seen that Maitland, in its current unoptimized form, induces up to several seconds delay when tracking a process which does not contain a signature the security analysis tool is aware of. In part, these delays arise due to the prototype's very restrictive page fault filtering policy. This policy requires that Maitland pass *all* guest-VM memory pages producing page faults up for analysis in the privileged VM. Figure 3, therefore, shows the worst-case performance in Maitland's most security restrictive mode of operation. Significant improvements to Maitland's overhead costs, i.e., the red versus green, are likely easily achievable through: i) by applying other contextual information available to Maitland within the guest VM's to minimize the number of page faults sent up for security analysis, and ii) quickly flagging out known good code through identifying its inherent signatures. The latter of course would require a set of such known good signatures to be developed, but this is a significantly easier process than creating malware signatures. Such optimizations though are the subject of future work.

It should also be noted that by the approach of (ii) above, Maitland's overhead costs would only be incurred by programs which are suspected of potentially being malicious; hence, no performance costs would need to be incurred by known good code once it has been detected as such. Additionally, as the observed unoptimized Maitland overhead is only in the neighborhood of a few seconds, this compares favorably with other known commercial cloud overheads. By comparing the blue and green bars it can be see that Maitland does indeed detect the known executable signatures early in the execution process, thereby, allowing appropriate defensive control actions to be undertaken before damage would occur.



(a) Pi_css5 Overhead (computation of 2^17 digits of Pi).



(b) Gzip Overhead (compression of a 10MB random file).

Fig. 3: Processing times for native execution (red) and with Maitland and security analysis enabled (blue), and with Maitland but without security analysis disabled (greed).

3) Experiment 3: Process Memory Size: Because Maitland traverses a process' page tree in order to collect its memory snapshot, Maitland's incurred overheads are also innately a function of the number of memory pages that need to be traversed. Figure 4 shows how Maitland's induced overheads increase with increasing number of process memory pages and, moreover, that this increase is non-linear, as per the Figure's log-log scale.

The observed dip at approximately 160 memory pages is likely an artifact of the specific settings contained within the Linux kernel's *slab allocator* (or *SA*) memory allocation optimization process[40]. Again, for the conducted tests, Figure

4 denotes the worst-case performance overhead for the unoptimized prototype Maitland implementation and approaches, such as the simple caching of unchanged pages, would likely be effective in significantly reducing these worst-case costs.

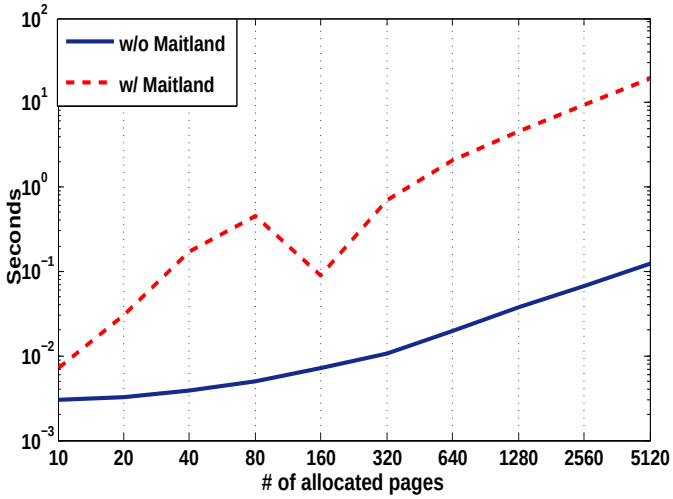


Fig. 4: Effects of guest VM process memory size on the observed Maitland overhead (Log-Log scale)

VI. Limitations

The above discussions clearly show that the prototype Maitland implementation has achieved the required VM introspection capabilities while retaining the desired light-weight nature. Moreover, the inclusion of a guest VM kernel component has been used to avoid the prior need for OS-patch level re-engineering. Within commercial cloud environments this guest VM requirement is reasonable given that the cloud provider also provides the authorized guest VM images. In more general contexts such a solution approach, obviously, would not apply. Cloud environments are unique in that they can be trivially structured to remove any advantages gained by the attacker through detecting the presence of VMs or the presence of the Maitland guest kernel component.

By the above performance numbers the prototype Maitland proof-of-concept implementation, obviously, retains substantial room for performance improvements. As discussed, these may be achieved through the appropriate use of memory page caching, through establishing a check-once process for unchanged pages, etc. Moreover, known techniques such as heuristic-based *stage completion approximation* [33], could be used to further lighten the overhead costs associated with detecting multi-round or multi-layer packing (or encryption) processes. It should also be noted that poor guesses by such heuristic approaches as to when unpacking or decryption processes may finish can lead to additional exploitable attack avenues. Hence, Maitland's performance was assessed in the worst-case when such heuristics cannot be applied.

Maitland also exists in the guest VM, as approximately 1000 lines of C code. Hence, it is small enough that formal methods could be applied to verify that Maitland itself does not introduce new security vulnerabilities. Again such a verification has been left as future work. As Maitland does not require hypervisor changes it expressly does not and, moreover, cannot introduce new security vulnerabilities to the hypervisor. Additionally, as Maitland's interfaces into guest VMs are constructed from standard file constructs these also innately cannot introduce new vulnerabilities. Of course, Maitland does introduce control paths of the privileged VMs security analysis tools back into the Maitland monitored guest VMs and their OSes. Hence, if these security analysis tools are themselves compromised these new control paths could be exploited. Of course, if a commercial cloud's security analyses themselves become compromised, then it is difficult to argue that any security continues to exist within the cloud, irrespective of any of these Maitland enabled control paths.

Interpreters and JIT compilers, of course, have become commonplace especially for web services. These produce similar behaviors to that of unpacking and decryption in that inputs are converted (or interpreted) into executable bytecodes. Maitland's initial prototype design and implementation does not seek to address such issues. However, because Maitland runs partially inside the guest VM OS kernel, it would be possible for it to interface directly with an interpreter, thereby, allowing a similar type of Maitland-enabled interface between the interpreter and the privileged VM security analysis tool. Such an interface could be constructed as a fairly straightforward Maitland extension. This, of course, raises additional issues as to whether the interpreter is itself trusted, which again are more feasible to address within commercial cloud environment.

A. Conclusion

Within this paper an approach to achieving lighter-weight VM introspection through exploiting the emerging shift towards paravirtualized VMs has been presented. Maitland, a prototype proof-of-concept implementation, has been developed and used to gain worst-case performance measures for this type of light-weight introspection approach. It was shown that Maitland's raw performance costs are on the order of a few extra seconds and, hence, within the range of other known commercial cloud overheads. Maitland's efficacy was demonstrated by using it to detect known binary signatures from packed executables and, importantly, in a manner that was packer independent. Hence, the enacted experiments can be trivially extended to cover multi-round and/or multi-layer unpacking or decryption now in common use to obfuscate malware executables. Maitland has also been designed to provide a simple interface through which any guest VM observable OS event can be propagated up and made visible to any privileged VM resident security analysis process, with the tested binary signature detection process being used as the exemplar. It was shown that Maitland also enables the passing back of VM security control actions, i.e., such as process termination, and it was discussed how this can be used to enable and foster a cloud-wide security service.

It can be seen that work remains to optimize the prototype implementation such that it would be real-world deployable. But, this work has shown that lighter-weight introspection is a viable approach and that it can be structured in a manner that does not required the hypervisor-level changes or per-OS patch re-engineering process of prior heavy-weight VM introspection techniques. Hence, Maitland arguably could provide a practical and pragmatic approach for providing VM introspection capabilities to modern commercial cloud platforms.

REFERENCES

[1] C. Economics, "2005 Malware Report: Executive Summary," 2006, http://www.computereconomics.com/article.cfm?id=1090.

[2] Gunter and Ollmann, "The evolution of commercial malware development kits and colour-by-numbers custom malware," *Computer Fraud and Security*, vol. 2008, no. 9, pp. 4 – 7, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1361372308701350

[3] J. Rutkowska and A. Tereshkin, "Bluepilling the Xen Hypervisor," *Black Hat USA*, 2008.

[4] S. King, P. Chen, C. Verbowski, H. Wang, and J. Lorch, "SubVirt: Implementing malware with virtual machines," *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pp. 314–327, 2006.

[5] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687

[6] C. Willems, T. Holz, and F. Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security and Privacy Magazine*, vol. 5, no. 2, pp. 32–39, Mar. 2007.

[7] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[8] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 128–138. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315262

[9] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether : Malware Analysis via Hardware Virtualization Extensions," *Analysis*, pp. 51–62, 2008.

[10] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," *2007 IEEE Symposium on Security and Privacy SP 07*, vol. 0, pp. 231–245, 2007.

[11] A. Stepan, "Improving proactive detection of packed malware." *Virus Bulletin*, vol. 1, no. March, 2006.

[12] F. Guo, P. Ferrie, and T. Chiueh, "A study of the packer problem and its solutions," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 98–115.

[13] P. Bustamante, "Mal(ware)formation statistics," 2007, http://research.pandasecurity.com/malwareformation-statistics.

[14] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee, "Eureka: A framework for enabling static malware analysis," *Computer Security-ESORICS 2008*, pp. 481–500, 2008.

[15] K. Timm, "Malware Validation Techniques," 2010, http://blogs.cisco.com/security/malware_validation_techniques/.

[16] G. Taha, "Counterattacking the packers," *McAfee Avert Labs, Aylesbury, UK*, 2007.

[17] Kaspersky, "Virus.Win32.Sality.bh," 2011, http://www.securelist.com/en/descriptions/15312802/Virus.Win32.Sality.bh\#doc1.

[18] J. Larimer, S. Researcher, and I. B. M. X-force, "An inside look at Stuxnet," *Agenda*, 2009.

[19] M. Fossi, G. Egan, K. Haley, E. Johnson, T. Mack, T. Adams, J. Blackbird, L. Mo King, D. Mazurek, D. McKinney, and P. Wood, "Symantec Internet Security Threat Report: Trend for 2010," *internet security threat*, vol. 16, no. April, 2011.

[20] M. Oberhumer, M.F. and J. ar, L., Reiser, "UPX: the Ultimate Packer for eXe- cutables (2007)," 2007, http://upx.sourceforge.net/.

[21] Oreans Technologies, "Themida," http://www.oreans.com/.

[22] ASPack Software, "ASPack," http://www.aspack.com/.

[23] D. M. Chess and S. R. White, "An Undetectable Computer Virus," in *Virus Bulletin Conference*. Citeseer, 2000.

[24] M. R. Chouchane and A. Lakhotia, "Using engine signature to detect metamorphic malware," *Proceedings of the 4th ACM workshop on Recurring malcode - WORM '06*, p. 73, 2006.

[25] D. Bruschi, L. Martignoni, and M. Monga, "Code Normalization for Self-Mutating Malware," *Ieee Security And Privacy*, vol. 5, no. 2, pp. 46–54, 2007.

[26] Fred and Cohen, "Computer viruses: Theory and experiments," *Computers and Security*, vol. 6, no. 1, pp. 22 – 35, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0167404887901222

[27] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 243–258.

[28] Amazon, "Amazon EC2 FAQs," 2011, http://aws.amazon.com/ec2/faqs/.

[29] StratusLab, "EC2, Xen and Paravirtualization," http://stratuslab.eu/doku.php/ec2_xen_and_paravirtualization.

[30] P. J. Salzman, M. Burian, and O. Pomerantz, "Talking to Device Files (writes and IOCTLs)," http://linux.die.net/lkmpg/x892.html.

[31] D. Chisnall, "The definitive guide to the xen hypervisor," *Journal of the Electrochemical Society*, vol. 129, p. 2865, 2007.

[32] J. Pfoh, C. Schneider, and C. Eckert, "A formal model for virtual machine introspection," *Conference on Computer and Communications Security*, pp. 1–10, 2009.

[33] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pp. 431–441, Dec. 2007.

[34] M. G. Kang, P. Poosankam, and H. Yin, "Renovo : A Hidden Code Extractor for Packed Executables," *October*, pp. 46–53, 2007.

[35] I. Corporation, "Intel 64 and IA-32 Architectures Software Developer s Manual Combined Volumes :," *Architecture*, no. October, 2011.

[36] D. Quist, V. Smith, and O. Computing, "Detecting the Presence of Virtual Machines Using the Local Data Table," *Offensive Computing-packetstormsecurity. org*, 2006.

[37] J. Rutkowska, "Redpill," 2004, http://www.invisiblethings.org/papers/redpill.html.

[38] T. Ooura, "Ooura's Mathematical Software Packages," 2006, http://www.kurims.kyoto-u.ac.jp/~ooura/.

[39] J.-l. Gailly and M. Adler, "The gzip home page," http://www.gzip.org/.

[40] B. Fitzgibbons, "The Linux slab allocator," 2000.