Argos: emulated hardware support to fingerprint zero-day attacks by means of dynamic data flow analysis

Georgios Portokalidis Asia Slowinska Herbert Bos

Department of Computer Science,
Faculteit der Exacte Wetenschappen,
Vrije Universiteit Amsterdam,
De Boelelaan 1081,
1081 HV Amsterdam, Netherlands
{porto, asia, herbertb}@few.vu.nl

Keywords: Operating Systems, Security, Honeypots

Abstract

As modern operating systems and software become larger and more complex, they are more likely to contain bugs, which may allow attackers to gain illegitimate access. A fast and reliable mechanism to discern and generate vaccines for such attacks is vital for the successful protection of networks and systems. In this paper we present *Argos*, a containment environment for worms as well as human orchestrated attacks. *Argos* is built upon a fast x86 emulator which tracks network data throughout execution to identify their invalid use as jump targets, function addresses, instructions, etc. Furthermore, system call policies disallow the use of network data as arguments to certain calls. When an attack is detected, we perform 'intelligent' process- or kernel-aware logging of the corresponding emulator state for further off-line processing. In addition, our own *forensics shellcode* is injected to gather information about the attacked process. By correlating the data logged by the emulator with the data collected from the network, the generation of accurate network intrusion detection signatures is made possible.

1 Introduction

The rate at which self-propagating attacks spread across the Internet has prompted a wealth of research in automated response systems. We have already encountered worms that spread across the Internet in as little as ten minutes, and researchers claim that even faster worms are just around the corner [1]. For such outbreaks human intervention is too slow and automated response systems are needed. Important criteria for such systems in practice are: (a) reliable *detection* of a wide variety of zero-day attacks, and (b) *cost-effective* deployment.

Existing automated response systems tend to incur a fairly large ratio of false positives in attack detection and use of signatures [2, 3, 4, 5, 6]. A large share of false positives violates the first criterion. Although these systems may play an important role in intrusion detection systems (IDS), it is problematic to apply them in fully automated response systems.

One approach that attempts to avoid false positives altogether is known as dynamic taint analysis. Briefly, untrusted data from the network is tagged and an alert is generated (only) if and when an exploit takes place, e.g., when data coming from the network are executed. This technique proves to be very reliable and to generate few, if any, false positives. It is used in current projects that can be categorised as (i) hardware-oriented full-system protection, and (ii) OS- and process-specific solutions in software. These are two rather different approaches, and each approach has important implications. For our purposes, the two most important representatives of these approaches are Minos [7] and Vigilante [8], respectively.

Minos relies on hardware for cost-effective deployment. Moreover, by looking at physical addresses only, it may *detect* certain exploits, such as a register spring attacks [9], but requires an awkward hack to determine where the attack originated [10]. Also, it cannot directly handle physical to virtual address translation at all.

In contrast, Vigilante represents a per-process solution that works with virtual addresses. Again, this is a design decision that limits its flexibility, as it is not able to handle DMA or memory mapping. Also, the issue of cost-effectiveness arises as Vigilante must in-

strument individual services and does not protect the OS kernel at all. Unfortunately, kernel attacks have become a reality and are expected to be more common in the future [11].

In this paper we describe *Argos* which explores another extreme in the design space for automated response systems. First, like Minos we offer whole-system protection in software by way of a modified x86 emulator which runs our own version of dynamic taint analysis [12]. In other words, we automatically protect any (unmodified) OS and all its processes, drivers, etc. Second, *Argos* takes into account complex memory operations, such as memory mapping and DMA (commonly ignored by other projects), and is at the same time quite capable of handling complex exploits (such as register springs). This is to a large extent due to our ability to handle both virtual and physical addresses. Third, buffer overflow and format string / code injection exploits trigger alerts that result in extensive logging of the emulator's state, which along with the network data comprise the most critical information for generating a NID signature. Fourth, while the system is OS- and application-neutral, when an attack is detected, we inject OS-specific forensics shellcode. In other words, we exploit the code under attack as the attack is happening to extract additional information about the attack.

We focus on attacks that are orchestrated remotely (like worms) and do not require user interaction. Approaches that take advantage of mis-configured security policies are not addressed. Even though such attacks constitute an ample security issue, they are beyond the scope of our work and require a different approach. Specifically, We focus on capturing attacks that exploit buffer overflows to inject code in order to gain control over a host. In our opinion, it is more useful to capture exploits, because a single exploit can be the carrier for different attacks. Additionally, exploits are less mutable than attack payloads and may be more easily caught even in the face of polymorphism.

Argos is designed as an 'advertised honeypot', i.e., a honeypot that runs real services and differs from normal honeypots in that we don't hide it. Rather, we actively link to it and 'advertise' its IP address in the hope of making it visible to attackers employing hitlists rather than random IP scanning to identify victims. The price we pay for this is that unlike conventional honeypots we expect to receive a fair amount of legitimate traffic (e.g., crawlers). On the other hand, since *Argos* is targeted as a honeypot, we do not require our solution to perform as well as unprotected systems. Nevertheless, it should be fast enough to run real services and have reasonable response time.

The remainder of this paper is organised as follows. Related work is discussed mainly throughout the text. In Section 2 we describe the design of *Argos*. Implementation details are discussed in Section 3. The system is evaluated in Section 4. Conclusions are in Section 5.

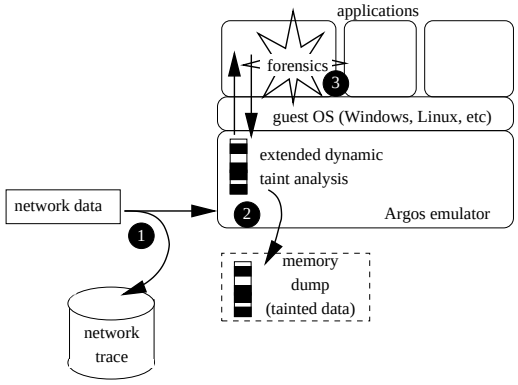

Figure 1: Argos: High-Level Overview

2 Design

An overview of the *Argos* architecture is shown in Figure 1. The full execution path consists of three main steps, indicated by the numbers in the figure which correspond to the circled numbers in this section. Incoming traffic is both logged in a trace database, and fed to the unmodified application/OS running on our emulator ①. In the emulator we employ dynamic taint analysis to detect when a vulnerability is exploited to alter an application's control flow ②. This is achieved by identifying illegal uses of possibly unsafe data such as the data received from the network [12]. There are three steps to accomplish this:

- tag data originating from an unsafe source as *tainted*;
- track *tainted* data during execution
- identify and prevent unsafe usage of *tainted* data;

In other words, data originating from the network is marked as tainted, whenever it is copied to memory or registers, the new location is tainted also, and whenever it is used, say, as a jump target, we raise an alarm. Thus far this is similar to approaches like [8] and [12]. As mentioned earlier, *Argos* differs from most existing projects in that we trace physical addresses rather than virtual addresses. As a result, the memory mapping problem disappears, because all virtual address space mappings of a certain page, refer to the same physical address.

When a violation is detected, an alarm is raised and *Argos* dumps all tainted blocks and some additional information to file, with markers specifying the address that triggered the violation, the memory area it was pointing to, etc. Since we have full access to the

machine, its registers and all its mappings, we are able to translate between physical and virtual addresses as needed. The dump therefore contains registers, physical memory blocks and specific virtual address. The log contains enough information not just for signature generation, but for, say, manual analysis as well.

In addition, we employ a novel technique to automate forensics on the code under attack. Recall that *Argos* is OS- and application-neutral, i.e., we are able to work out-of-the-box with any OS and application on the IA32 instruction set architecture (no modification or recompilation required). When an attack is detected, we may not even know which process is causing the alarm. To unearth additional information about the application (e.g., process identifier, executable name, open files and sockets, etc.), we inject our own shellcode to perform forensics ③. In other words, we 'exploit' the code under attack with our own shellcode. To the best of our knowledge, we are the first to employ the means of attack (shellcode) for defensive purposes.

3 Implementation

Argos extends the *Qemu* [13] hardware emulator by providing it with the means to taint and track memory, and generate memory footprints in case of a detected violation. *Qemu* is a fast and portable dynamic translator that emulates multiple architectures such as `x86`, `x86_64`, `POWER-PC64`, etc. Unlike other emulators such as Bochs [14], *Qemu* is not an interpreter. Rather, entire blocks of instructions are translated and cached so the process is not repeated if the same instructions are executed again. Furthermore, instead of providing the software equivalent of a hardware system, *Qemu* employs various optimisations to improve performance. As a result, *Qemu* is significantly faster than most emulators.

Our implementation extends *Qemu*'s Pentium architecture. In the remainder of this paper, it will be referred to simply as the `x86` architecture. For the sake of clarity we will also use the terms guest and host to distinguish between the emulated system and the system hosting *Qemu*.

We divide our implementation of *Argos* in two parts. The first contains our extended dynamic taint analysis which we used both to secure *Qemu* and to enable it to issue alerts whenever it identifies an attack. The second part covers the extraction of critical information from the emulator and the OS.

3.1 Extended Dynamic Taint Analysis

The dynamic taint analysis in *Argos* resembles that of other projects. However, there are important differences. In this section we discuss the implementation details.

3.1.1 Tagging

An important implementation decision in taint analysis concerns the granularity of the tagging. In principle, one could tag data blocks as small as a single bit, up to chunks of 4 KB or larger. We opted for variable granularity; per byte tagging of physical memory, while at the same time using a single tag for each CPU register. Per byte tagging of memory does not incur any additional computational costs i.e. over per double word tagging, and provides higher accuracy. On the other hand, per byte tagging of registers would introduce increased complexity in register operations, which is unacceptable. It is worth noting that altering *Argos* to employ a different granularity is trivial. For reasons of performance and to facilitate the process of forensics at a later stage, the nature of the memory and register tags is also different.

Register tagging There are eight general purpose registers in the `x86` architecture [15], and we allocate a 4 B tag for each of them. The tag is used to store the physical memory address from where the contents of the register originate. Segment registers and the instruction pointer register (`EIP`) are not tagged and are always considered *untainted*. Since they can only be altered implicitly and because of their role, they belong to the protected elements of the system. The `EFLAGS` register is also not tagged and is considered *untainted*, because it is frequently affected by operations involving untrusted data, and tagging it would make it impossible to differentiate between malicious and benevolent sources. By default, `MMX` and `FPU` registers are treated similarly, although *Argos* is *able* to tag them if required. We implemented tagging for these registers as an option only, since they are involved in very specific operations that are rarely, if ever, involved in attacks. For the sake of performance, we ignore them by default.

Memory tagging Since we do not store any additional data for physical memory tags, a binary flag for tagging would suffice. Nevertheless, one could also use a byte flag increasing memory consumption in exchange for performance gains. This might seem costly, but recall that we tag *physical* rather than virtual memory. While virtual memory space may be huge (e.g., 2^{64} on 64-bit machines) the same is not true for physical memory, which is commonly on the order of 512 MB - 1GB. Moreover, the guest's 'physical' RAM need not correspond to the physical memory at the host, so the cost in hardware resources can be kept fairly small. The scheme to be used can be configured at compile time. Following, we will discuss the two tagging schemes in more detail.

A *bitmap* is a large array, where every byte corresponds to 8 bytes in memory. The index idx of

any physical memory address $paddr$ in the bitmap can be calculated by first shifting the address right by 3 ($idx = paddr \gg 3$) to locate the byte containing the bit flag ($map[idx]$). The individual bit flag is retrieved by using the lower 3 bits of $paddr$ ($b = map[idx] \bigoplus (0x01 \ll (paddr \bigoplus 0x07))$). The $size$ of the bitmap is an eighth of the guest's total addressable physical memory $RAMSZ$ ($size = \frac{RAMSZ}{8}$), i.e. the bitmap for a guest system of 512 MB would be 64 MB.

Similarly, a *bytemap* is also a large array, where each byte corresponds to a byte in memory. The physical address $paddr$ of each byte is also the index idx in the bytemap. Its total $size$ is equal to the guest's total addressable physical memory $RAMSZ$ ($size = RAMSZ$).

Finally, incoming network data are marked as *tainted*. Since the entire process does not involve OS participation the tagging is performed by the virtual `NE2000` NIC emulated by *Qemu*. OSs communicate with peripherals in two ways: port I/O and memory mapped I/O. *Qemu*'s virtual NIC though, supports only port I/O, which in `x86` architectures is performed using instructions `IN` and `OUT`. By instrumenting these instructions the registers loaded with data from the `NE2000` are tagged as *tainted* while all other port I/O operations result in clearing the destination register's tag.

3.1.2 Tracking

Qemu translates all guest instructions to host native instructions by dynamically linking blocks of functions that implement the corresponding operations. Tracking *tainted* data involves instrumenting these functions to manipulate the tags, as data are moved around or altered. Besides registers and memory locations, available instruction operands include immediate values, which we consider to be *untainted*. We have classified instrumented functions in the following categories:

- *2 or 3 operand* `ALU` *operations;* these are the most common operations and include `ADD`, `SUB`, `AND`, `XOR`, etc. If the destination operands are not *tainted*, they result in copying the source operands tags to the destination operands tags.
- *Data move operations;* these operations move data from register to register, copying the source's tag to the destination's tag.
- *Single register operations;* shift and rotate ops belong to this category. The tag of the register is either preserved as it is, or cleared when data are altered in a complex way that would be extremely hard for an attacker to exploit (e.g. `BCD`, rotate and bit set instructions [16] are considered to 'sanitise' data).
- *Memory related operations;* all `LOAD`, `STORE`, `PUSH` and `POP` operations belong here. These operations retrieve or store the tags from or to memory respectively.
- `FPU`, `MMX`, *or* `SSE` *operations;* as explained above, these are ignored by default (tagging of the corresponding registers is optional in *Argos*), unless their result is stored in one of the registers we track or to memory. In these cases, the destination is cleared. More advanced instructions such as `SSE2` and `3DNow!` are not supported by *Qemu*.
- *Operations that do not directly alter registers or memory;* some of these ops are `NOP`, `JMP`, etc. For most of these we do not have to add any instrumentation code for tracking data, but for identifying their illegal use instead, as we describe in the following section.

Fortunately, we do not have to worry about special instruction uses such as `xor eax,eax` or `sub eax, eax`. These are used in abundance in `x86`'s to set a register to zero, because unlike RISC there is no zero register available. *Qemu* makes sure to translate these as a separate function that moves zero to the target register. When this function is compiled it follows the native architecture's idiom of zeroing a register.

Modern systems provide a mechanism for peripherals to write directly to memory without consuming CPU cycles, namely direct memory access (DMA). When using DMA OSs instead of reading small chunks of data from peripherals they allocate a larger area of memory and send its address to the peripheral, which in turn writes data directly in that area without occupying the CPU. *Qemu* emulates DMA for components such as the hard disk. Whenever a DMA write to memory is performed in *Argos*, it is intercepted and the corresponding memory tags are cleared.

3.1.3 Preventing Invalid Uses of Tainted Data

Most of the observed attacks today gain control over a host by redirecting control to instructions supplied by the attacker (e.g., shellcode), or to already available code by carefully manipulating arguments (return to `libc`). For these attacks to succeed the instruction pointer of the host must be loaded with a value supplied by the attacker. In the `x86` architecture, the instruction pointer register `EIP` is loaded by the following instructions: `call`, `ret`, and `jmp`. By instrumenting these instructions to make sure that a *tainted* value is not loaded in `EIP`, we manage to identify all attacks employing such methods. Optionally, we can also check whether a *tainted* value is being loaded on

model specific registers (MSR) or segment registers, but so far we have not encountered such attacks and we are not aware of their existence.

While these measures capture a broad category of exploits, they alone are not sufficient. For instance, they are unable to deal with format string vulnerabilities, which allow an attacker to overwrite any memory location with arbitrary data. These attacks do not directly overwrite critical values with network data, and might remain undetected. Therefore, we have extended dynamic taint analysis to also scan for code-injection attacks that would not be captured otherwise. This is easily accomplished by checking that the memory location loaded on EIP is not *tainted*.

Finally, to address attacks that are based solely on altering arguments of critical functions such as system calls, we have instrumented *Qemu* to check when arguments supplied to system calls like execve() are *tainted*. To reliably implement this functionality we require a hint about the OS being run on *Argos*, since OSs use different system calls. The current version of *Argos* supports this feature solely for the Linux OS.

3.2 Attack Logging

In this section we explain the way *Argos* logs the detected attacks, as well as advanced forensics.

3.2.1 Extracting Data

An identified attack can become an asset for the entire network security community if we generate a signature to successfully block it at the network level. To allow this, *Argos* exports the contents of 'interesting' memory areas in the guest for off-line processing. To reduce the amount of exported data we dynamically determine whether the attack occurred in user- or kernel-space. This is achieved by retrieving the processor's privilege ring bits from *Qemu*'s hidden flags register. The kernel is always running on privileged ring 0, so we can distinguish processes from the kernel by looking at the ring in which we are running.

Additionally, every process is sharing its virtual address space with the kernel. OSs accomplish this by splitting the address space. In the case of Linux a 3:1 split is used, meaning that three quarters of the virtual address space are given to the process while one quarter is assigned to the kernel. Windows on the other hand is using a 2:2 split. The user/kernel space split is predefined in most OS configurations, so we are able to use static values as long as we know which OS is being run. We take advantage of this information to dump only relevant data.

To determine which physical memory pages are of interest and need to be logged, we traverse the page directory installed on the processor. In x86 architectures the physical memory address of the active page directory is stored in control register 3 (CR3). Note

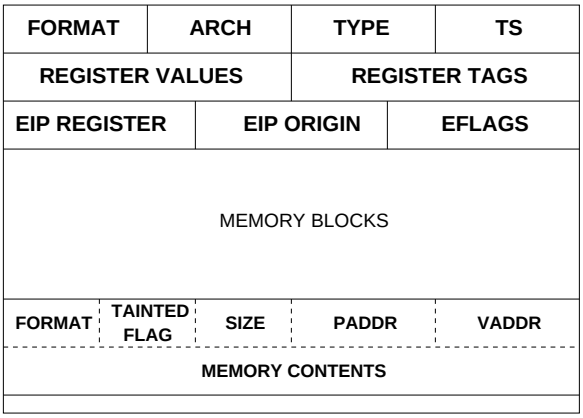

Figure 2: Memory dump format

that because we traverse the virtual address space of processes, physical pages mapped to multiple virtual addresses will be logged multiple times (one for each mapping).

By locating all the physical pages accessible to the process / kernel, and making sure that we do not cross the user / kernel space split, we dump all *tainted* memory areas as well as the physical page pointed to by EIP regardless of its tags state. The structure of the dumped data is shown in Figure 2. For each detected attack the following information is exported: the log's format (FORMAT), the guest architecture (ARCH could be i386 or x86_64), the type of the attack (TYPE), the timestamp (TS), register contents and tags (including EIP and its origin), the EFLAGS register, and finally memory contents in blocks. Each memory block is preceded by the following header: the block's format (FORMAT), a *tainted* flag, the size of the block in bytes, and the physical (PADDR) and virtual (VADDR) address of the block. The actual contents of the memory block are written next. When all blocks have been written, the end of the dump is indicated by a memory block header containing only zeroes.

All of the above are logged in a file named 'argos.csi.RID', where RID is a random ID that will be also used in advanced forensics discussed in the following section.

The data extracted from *Argos* serve for more than signature generation. By logging all potentially 'interesting' data, thorough analysis of the attack is made possible helping security experts understand new attacks.

3.2.2 Advanced Forensics

An intrinsic characteristic of *Argos* is that it is process agnostic. This presents us with the problem of identifying the target of an attack. Discovering the victim process, provides valuable information that can be used to locate vulnerable hosts, and assist in signa-

ture generation. To overcome this obstacle, we came up with a novel idea that enables us to execute code in the process's address space, thus permitting us to gather information about it.

Currently, most attacks hijack processes by injecting assembly code (shellcode) and diverting control flow to its beginning. Inspired by the above, we inject our own shellcode into a process's virtual address space. After detecting an attack and logging state, we place forensics shellcode *directly* into the process's virtual address space. The location where the code is injected is crucial, and after various experiments we chose the last `text` segment page at the beginning of the address space. Placing the code in the `text` segment is important to guarantee that it will not be overwritten by the process, since it is read-only. It also increases the probability that we will not overwrite any critical process data. Having the shellcode in place we then point `EIP` to its beginning to commence execution.

As an example, we implemented shellcode that extracts the `PID` of the victim process, and transmits it over a TCP connection along with the `RID` generated previously. The information is transmitted to a process running at the guest, and the code then enters a loop that forces it to sleep forever to ensure that while it does not terminate, it remains dormant. At the other end, an information gathering process at the guest receives the `PID` and uses it to extract information about the given process by the OS. Finally, this information is transmitted to the host, where it is stored.

The forensics process retrieves information about the attacked process by running `netstat`, or if that is not available `OpenPorts` [17]. The above tools offer both the name of the process, as well as all the associated ports. Currently, forensics are available for both Linux and win32 systems. In the future, we envision extracting the same or more information without employing a 3rd process at the guest.

4 Evaluation

We evaluate *Argos* along two dimensions: performance and effectiveness. While performance is not critical for a honeypot, it needs to be fast enough to generate signatures in a timely fashion.

4.1 Performance

For realistic performance measurements we compare the speed of code running on *Argos* with that of code running without emulation. We do this for a variety of realistic benchmarks, i.e., benchmarks that are also used in real-life to compare PC performance. Note that while this is an honest way of showing the slowdown incurred by *Argos*, it is not necessarily the most relevant measure. After all, we do not use *Argos* as a desktop and in practice hardly care whether results appear much less quickly than they would without emulation. The only moment when slowdown becomes an issue is when attackers decide to shun slow hosts, because it might be a honeypot. To the best of our knowledge such worms do not exist in practice.

Performance evaluation was carried out by comparing the observed slowdown at guests running on top of various configurations of *Argos* and unmodified *Qemu*, with the original host. The host used during these experiments was an AMD Athlon^{TM} XP 2800 at 2 GHz with 512 KB of L2 cache, 1 GB of RAM and 2 IDE UDMA-5 hard disks, running Gentoo Linux with kernel 2.6.12.5. The guest OS ran SlackWare Linux 10.1 with kernel 2.4.29, on top of *Qemu* 0.7.2 and *Argos*. To ameliorate the guest's disk I/O performance, we did not use a file as a hard disk image, but instead dedicated one of the hard disks.

To quantify the observed slowdown we used `bunzip2` and `apache`. `bunzip2` is a very CPU intensive UNIX decompression utility. We used it to decompress the Linux kernel v2.6.13 source code (approx. 38 MB) and measured its execution time using another UNIX utility `time`. `Apache`, on the other hand, is a popular web server that we chose because it enables us to test the performance of a network service. We measured its throughput in terms of maximum processed requests per second using the `httperf` HTTP performance tool. `httperf` is able to generate high rates of single file requests to determine a web server's maximum capacity.

In addition to the above, we used BYTE magazine's UNIX benchmark. This benchmark, `nbench` for brevity, executes various CPU intensive tests to produce three indexes. Each index corresponds to the CPU's integer, float and memory operations and represents how it compares with an AMD K6^{TM} at 233 MHz.

Figure 3 shows the results of the evaluation. We tested the benchmark applications at the host, at guests running over the original *Qemu*, and at different configurations of *Argos*: using a bytemap, and using a bytemap with code-injection detection enabled. These are indicated in the figure as Vanilla QEMU, Argos-B, and ARGOS-B-CI respectively. The y-axis represents how many times slower a test was, compared with the same test without emulation. The x-axis shows the 2 applications tested along with the 3 indexes reported by `nbench`. Each colour in the graph is a configuration tested, which from top to bottom are: unmodified *Qemu*, *Argos* using a bytemap for memory tagging, and the same with code-injection detection enabled.

Even in the fastest configuration, *Argos* is at least 16 times slower than the host. Most of the overhead, however, is incurred by *Qemu* itself. *Argos* with all the additional instrumentation is at most 2 times

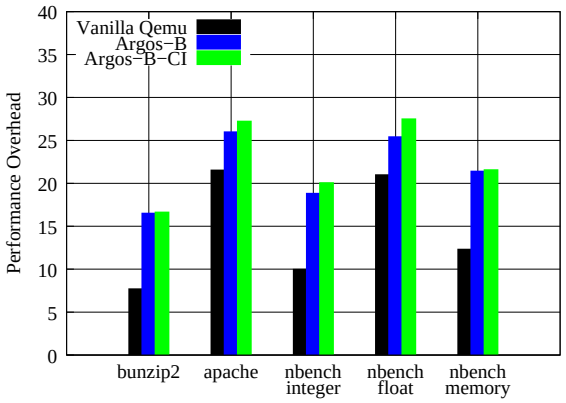

Figure 3: Performance Benchmarks

slower than vanilla *Qemu*. In the case of `apache` and float operations specifically, there is only an 18% overhead. This is explained by the lack of a real network interface, and a hardware FPU in the emulator, which incurs most of the overhead. In addition, we emphasise that we have not used any of the optimisation modules available for *Qemu*. These modules speed up the emulator to a performance of roughly half that of the native system. While it is likely that we will not quite achieve an equally large speed-up, we are confident that much optimisation is possible.

Moreover, even though the performance penalty is large, personal experience with *Argos* has shown us that it is tolerable. Even when executing graphics-intensive tasks, the machine offers decent usability to human operators who use it as a desktop machine. Moreover, we should bear in mind that *Argos* was not designed as a desktop system, but as a platform for hosting advertised *honeypots*. Performance is not our main concern. Still, we have plans to introduce novelties that will further improve performance in future versions of *Argos*.

4.2 Effectiveness

To determine how effective *Argos* is in capturing attacks, we launched multiple exploits against both Windows and Linux operating systems running on top of it. For the Windows 2000 OS, we used the Metasploit framework [18] which provides ready-to-use exploits, along with a convenient way to launch them. We tested *all* exploits for which we were able to obtain the software. In particular, all the attacks were performed against vulnerabilities in software available with the professional version of the OS, with the exception of the War-FTPD ftp server which is third-party software. While we have also successfully run other OSs on *Argos* (e.g., Windows XP), we have only just started its evaluation. For the Linux OS, we crafted two applications containing a stack and a heap buffer overflow respectively and also used `nbSMTP`, an SMTP client that contains a remote format string vulnerability that we attacked using a publicly available exploit.

Vulnerability	**OS**
Apache Chunked Encoding Overflow	Windows
Microsoft IIS ISAPI .printer Extension Host Header Overflow	Windows
Microsoft Windows WebDav ntdll.dll Overflow	Windows
Microsoft FrontPage Server Extensions Debug Overflow	Windows
Microsoft LSASS MS04-011 Overflow	Windows
Microsoft Windows PnP Service Remote Overflow	Windows
Microsoft ASN.1 Library Bitstring Heap Overflow	Windows
Microsoft Windows Message Queueing Remote Overflow	Windows
Microsoft Windows RPC DCOM Interface Overflow	Windows
War-FTPD 1.65 USER Overflow	Windows
nbSMTP v0.99 remote format string exploit	Linux
Custom Stack Overflow	Linux
Custom Heap Corruption Overflow	Linux

Table 1: Exploits Captured by Argos

A list of the tested exploits along with the underlying OS is shown in table 1. For Windows, we have only listed fairly well-known exploits. All exploits were successfully captured by *Argos* and the attacked processes were consequently stopped to prevent the exploit payloads from executing. In addition, our forensics shellcode executed successfully, providing us with the process's name and PID.

Finally, we should mention that during the performance evaluation, as well as the preparation of attacks, *Argos* did not generate any false alarms about an attack. A low number of false positives is crucial for automated response systems. Even though the results do not undeniably prove that *Argos* will *never* generate false positives, considering the large number of exploits tested, it may serve as an indication that *Argos* is fairly reliable.

5 Conclusion

In this paper we have discussed an extreme in the design space for automated intrusion detection and response system: a software-only whole-system solution based on an `x86` emulator that uses dynamic taint analysis to detect exploits and protects unmodified operating systems, processes, etc. By choosing a vantage point that incorporates attractive properties from both the hardware level (e.g., awareness of

physical addresses, memory mapping and DMA) and also the higher-levels (virtual addresses, per-process forensics), we believe our approach is able to meet the demands of automated response systems better than existing solutions.

The system exports the tainted memory blocks and additional information as soon as an attack is detected, at which point it injects forensics shellcode into the code under attack to extract additional information (e.g., executable name and process identifier). Performance without employing any of the emulator's optimisation modules is significantly slower than code running without the emulator. Even so, as our intended application domain is (advertised) honeypots, we believe the overhead is acceptable. More importantly, the system proved to be effective and was used to capture and fingerprint a range of real exploits.

References

[1] Vern Paxson Stuart Staniford and Nicholas Weaver. How to 0wn the internet in your spare time. In *Proc. of the 11th USENIX Security Symposium*, 2002.

[2] Matthew M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In *Proc. of ACSAC Security Conference*, Las Vegas, Nevada, 2002.

[3] S. Singh, C. Estan, G. Varghese and S. Savage. Automated worm fingerprinting. In *In Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45–60, 2004.

[4] Kim Hyang-Ah and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *In Proc. of the 13th USENIX Security Symposium*, 2004.

[5] D. Dagonand, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine and Henry Owen. HoneyStat: Local worm detection using honeypots. In *In Proc. of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.

[6] Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.

[7] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *In Proc. of the 37th annual International Symposium on Microarchitecture*, pages 221–232, 2004.

[8] M. Costa, J. Crowcroft, M. Castro, A Rowstron, L. Zhou, L. Zhang and P. Barham. Vigilante: End-to-end containment of internet worms. In *In Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005.

[9] Dark Spyrit. Win32 buffer overows (location, exploitation, and prevention). Phrack 55, 1999.

[10] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Intrusion and Malware Detection and Vulnerability Assessment: Second International Conference (DIMVA05)*, Vienna, Austria, July 2005.

[11] Barnaby Jack. Remote windows kernel exploitation - step into the ring 0. eEye Digital Security Whitepaper, `www.eeye.com/~data/publish/whitepapers/research/OT20050205.FILE.pdf`, 2005.

[12] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[13] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *In Proc. of the USENIX Annual Technical Conference*, pages 41–46, April 2005.

[14] Kevin Lawton et al. Bochs ia-32 emulator project. `http://bochs.sourceforge.net`.

[15] *Basic Architecture*, volume 1 of *Intel Architecture Software Developer's Manual*. Intel Corporation, 1997.

[16] *Instruction Reference*, volume 2 of *Intel Architecture Software Developer's Manual*. Intel Corporation, 1997.

[17] Diamondcs openports. `http://www.diamondcs.com.au/openports/`.

[18] Metasploit project. `http://www.metasploit.com/`.