Stealth Measurements for Cheat Detection in On-line Games

Wu-chang Feng
Portland State University

Ed Kaiser
Portland State University

Travis Schluessler
Intel

ABSTRACT

As a result of physically owning the client machine, cheaters in network games currently have the upper-hand when it comes to avoiding detection by anti-cheat software. To address this problem and turn the tables on cheaters, this paper examines an approach for cheat detection based on the use of stealth measurements via tamper-resistant hardware. To support this approach, we examine a range of cheat methods and a number of measurements that such hardware could perform to detect them.

1. INTRODUCTION

When it comes to cheating in on-line games, the adversary holds all the cards. With complete control over the client, cheaters can hide their presence by modifying the operating system, disabling or spoofing anti-cheat software, and even removing their cheat from the system just before anti-cheat software runs. On top of these techniques, the cheater can employ anti-debugging, obfuscation, and evasion techniques that prevent game developers from deploying appropriate counter-measures in anti-cheat systems. This paper examines a new approach for tackling the cheating problem by employing hardware-based, stealth measurements. In this approach, a tamper-resistant processor resident on a client and isolated from the system's primary processor is used to perform measurements "underneath" the host's software stack. As a result of the processor's isolation, it is much more difficult for a cheater to disable or modify the measurements, or determine what is being measured and when.

2. MOTIVATION

Since cheaters own the machines they play on, they can currently tamper with any part of the operating system, the target game, and the anti-cheat software in order to both cheat and evade detection. The crux of our approach is to utilize a hardware component within the system as a means for providing tamper-resistant "ground-truth" measurements of a remote game system. Specifically, such hardware must be isolated from the host processor, cannot be affected by the system owner, must run only code signed by a trusted third party, and must have sufficient access to the running system. Given these attributes, the hardware component can perform measurements that enable an observer to determine if cheat software has compromised a game.

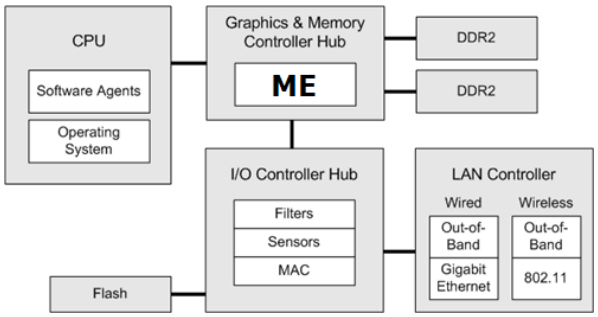

Figure 1: Functional location of one instance of Intel AMT's Manageability Engine (ME).

As an example of one possible instantiation, Figure 1 shows the current architecture of Intel's Active Management Technology platform [7] that has been used to detect rootkits in operating systems [3] and automation cheats in on-line games [10]. As the figure shows, AMT's trusted processor, the Manageability Engine (ME), has access to the contents of physical memory, all network traffic going to and from the host, and all I/O going to and from peripherals. One important limitation of this instance of the ME is that its placement outside of the CPU precludes measurement of important hardware registers such as the Interrupt Descriptor Table Register, the debug registers (DR0-DR7), the control registers (CR0-CR4), and the instruction pointer. As a result, the focus in this paper will go beyond the measurements that current platforms can perform and examine what measurements future trusted hardware platforms *should* perform in order to reliably detect cheating.

There are two significant advantages that hardware-based measurements provide against game hackers writing cheat software: they do not reveal *what* is being measured and they do not reveal *when* the measurements are being performed. As a result, the developers of cheat software do not know whether their cheats will escape undetected and are unable to determine when measurements are taken so that they cannot hide their cheats. This fear of the unknown has been shown to be an effective deterrent for cheats involving World of Warcraft, the most popular MMORPG

NetGames '08 Worcester, Massachusetts USA

being played today [5]. Note that while a stealth, hardware-based approach is necessary to gain the upper-hand against cheaters, it clearly is a double-edged sword that prevents users from knowing that their privacy is not being compromised [4]. Key to any approach is the ability to ensure that while the exact measurement being made is unknown, its scope is. That is, one must ensure that measurements don't extend to sensitive information outside the game.

3. METHODS

To understand what measurements could reliably reveal the presence of cheat software, it is important to survey methods that are currently being used to cheat. Note that many methods are used in conjunction with each other to implement a particular cheat.

3.1 Authorized, Automated Data Read

This method automates the collection of information that is presented and available to the player. Such cheats typically use legitimate APIs to learn game data without directly interacting with the game process. Because the APIs serve legitimate purposes (mostly for accessibility), their abuse is difficult to detect. A common technique is:

- Using the Graphics Device Interface API (specifically `BitBlt()` and `GetPixel()`) to dump pixel information from the screen and discern game state.

This method is typically used when implementing bots that automate actions, like fishing bots (such as FishBuddy) for World of Warcraft.

3.2 Unauthorized Data Read

This method, also known as information exposure, accesses hidden game data that should not be revealed to the player. Specific techniques include:

- Using a packet sniffer to extract game data from unencrypted network traffic.
- Using APIs (such as `ReadProcessMemory()`) to remotely read the memory of the game process.
- Using an injected or hijacked thread (see Section 3.5) to directly access memory from within the game process.

Cheats that employ this method include map-hacks that reveal the location of enemy units behind the "fog of war" in Warcraft III, wall-hacks that reveal the exact locations of enemy players behind walls in Counter-Strike, and Kick-Ass Map that reveals mob locations beyond the player's view in World of Warcraft.

3.3 Unauthorized Data Write

One of the simplest methods to alter game behavior is to directly modify data within the game process. By changing the data that the game uses, cheaters can gain abilities that their opponents cannot. Techniques for performing unauthorized data writes include:

- Using APIs (like `WriteProcessMemory()`) to remotely write to the memory of the game process.
- Using an injected or hijacked thread (see Section 3.5) to modify memory from within the game process.

Cheats that use *static* data writes include modifying the gravity constant so that players can climb walls in World of Warcraft and modifying memory-mapped wall textures to make them transparent in Counter-Strike. While these examples are easy to detect since they modify data to invalid values for the duration of the cheat, other cheats *dynamically* toggle data between valid values in an illegitimate way. For example, a cheat for Battlefield 2 continuously changes the team a player is on in order to trick the game client into revealing enemy locations via the radar.

Cheat Engine [1] is a tool that uses this method to locate and dynamically modify game data. In particular, it modifies the player's coordinates and direction to implement teleport cheats, Z-hack cheats (where the player is kept a fixed distance from the opponent during battle), and direction cheats (where the player is made to face the opponent at all times). This method is also used to illegally modify a character's attributes such as experience or health level.

3.4 Code Injection

Many cheats change the operation of the game by altering the game code or running their own code within the game process. To achieve this, cheaters can inject their code into the process during the loading of the game or from an external process while the game is running. There are a myriad of ways that code can be injected including:

- Using `WriteProcessMemory()` to *hot patch* (i.e. overwrite) game code to implement new functionality or make particular game operations always or never happen. For example, one can disable flash grenade blinding effects in Counter-Strike by changing jump instructions that call the flash effect into NOPs. Hot patching is feasible if the changes can be done within the size constraints of the original function.
- Using `WriteProcessMemory()` to write to *code-caves*, pockets of allocated but unused memory between existing game functions. The use of code-caves provides some stealth if the anti-cheat only scans the game's original code locations.
- Allocating new memory using `VirtualAllocEx()` and writing to it using `WriteProcessMemory()`. This is not stealthy, but facilitates injecting an arbitrary amount of cheat code into the game process.
- Loading a Dynamic Link Library (DLL) containing the cheat payload by either using `LoadLibrary()`, by hooking `LoadLibrary()` as it is called for other game DLLs, or by modifying registry entries such as `AppInit_DLL` to have the cheat DLL loaded automatically with the game. Tools that support this technique include Winject and INJLIB.

Cheats that inject code are prevalent and include BlackOmega for Battlefield 2, Ecstatic for Counter-Strike, and HL2 Hook for Half-Life 2.

3.5 Thread Manipulation

Once cheat code is injected into the game, it must be executed. The most common techniques involve manipulating threads within the game process by:

- Using *detours* (or *trampolines*) to temporarily hijack an existing game thread [6]. The detour redirects game

function calls to injected cheat code by hot patching a handful of bytes at the beginning of game functions. The overwritten bytes include a jump instruction that points to previously injected cheat code. Depending on the intention, some detours will execute the game function after executing the cheat code.

- Injecting a new thread via `CreateRemoteThread()` to execute cheat code alongside the game's normal threads.

3.6 Direct Function Calls

Many cheats change the operation of the game by directly calling operating system or game functions as needed for the desired behavior. This is especially true for automation bots which take input from authorized or unauthorized reads (Sections 3.1 & 3.2), make decisions, and then directly call game code to make actions. Techniques include:

- Using I/O APIs (`keybd_event()` or `mouse_event()`) directly instead of from the keyboard/mouse interrupt handler.
- Using an injected or hijacked thread (see Section 3.5) to directly call functions from within the game process.

Use of direct function calls is prevalent in bots and is a key component of Hoglund's World of Warcraft Implant bot [5].

3.7 Function Pointer Hooks

Since function pointers are part of a program's data they are typically unmeasured by static cheat detection measurements. Cheaters can take advantage of this by loading a cheat into memory and modifying function pointers in the game or operating system in order to invoke the cheat. Function pointers are prevalent in any running process whether it is within the game's code, the libraries it uses or the operating system. By overwriting function pointers in order to execute injected cheat code rather than overwriting the functions themselves, this method allows the game to still pass integrity checks that only examine the game's code. Examples include:

- Return address hooks that modify pointers stored in the stack so that functions return to injected code rather than their caller. Such a method is the basis for return-to-libc attacks [11].
- Overwriting function pointers in game code that implement run-time binding of operations or jump table implementations of switch statements in C and C++.
- Import Address Table (IAT) hooks that replace the game process's table of function pointers for functions exported by loaded DLLs.
- Interrupt Descriptor Table (IDT) hooks that replace entries in the table which stores interrupt handler function pointers.
- System Service Dispatch Table (SSDT) hooks that replace pointers in the system call table.
- I/O Request Packet (IRP) Function Table hooks that replace functions in the kernel's I/O processing chain.
- Structured Exception Handler (SEH) hooks that replace exception handler pointers on the stack with addresses for cheat code.
- Windows message hooks that use the Windows API to hook function handlers across all running processes via `SetWindowsHookEx()`.

Function pointer hooks are prevalent in bots across all game genres such as HL2Hook/CSHook and speed hacks implemented using Cheat Engine [1].

3.8 External Processes

In this method, the cheater employs an external process that modifies or tampers with the game process. Instances of this method include:

- Using standard APIs described previously to access and tamper with the game process (`ReadProcessMemory()`, `WriteProcessMemory()`, `CreateRemoteThread()`, and `VirtualAllocEx()`).
- Sending Windows messages such as mouse and keyboard events to the game process via APIs like `SendMessage()`. For example, in first-person shooters, automated events are sent in order to perform recoil suppression (e.g. `WM_MOUSEMOVE`).
- Using `DebugActiveProcess()` to attach to the game process as a debugger and completely control its execution. This technique can be used to either change the game code itself or to hijack game process threads to load cheat code from libraries.

Examples include recoil suppression cheats in first-person shooters that inject mouse events from a remote process and any cheats that use the debugger interface to gain control of the game process.

3.9 File Replacement

In this method, the cheater modifies the game binary, the game's data files, the libraries the game uses, and/or kernel modules on disk. While this method has been used in the past, file integrity checks by anti-cheat software have rendered this less popular. Specific examples include wall hacks in first-person shooters that replace game texture files with transparent alternatives.

3.10 Hardware Facilities

Because of the difficulty in correctly measuring hardware state from software, cheats that tamper with hardware features are some of the most difficult to detect. Such cheats span a wide-range and include:

- Tampering with the Interrupt Descriptor Table Register (IDTR) of the CPU which stores the pointer to where the Interrupt Descriptor Table (IDT) resides. As described previously, cheats can modify the IDT in order to replace interrupt handler function pointers with pointers to cheat code. Since checking the integrity of the pointers in this table can detect such tampering, a stealthier method would be to keep the original table intact, but instead change the IDTR so that it points to a completely different table containing pointers to cheat code.
- Processors typically support hardware debug registers that stop execution and cause an exception to occur whenever particular code locations are reached or when

particular memory locations are accessed. These exceptions allow the debugger to run. By using this facility, cheat software can hijack execution without explicitly injecting debugger interrupt instructions into the original game code.

- To gain access to memory, cheats can tamper with the memory management subsystem of a processor including its control and segment registers. For example, tampering with IA-32 control registers (CR0-CR3) can allow a cheat to modify read-only code pages and to hide the memory pages where their code resides.

- Model Specific Registers (MSRs) can also be used to tamper with the game and operating system in a variety of ways. One specific example is the SYSENTER_EIP_MSR register on IA-32 CPUs that holds the address of the "fast" system call function. By modifying this register, a cheat can hook essential system calls underneath a game.

- Another way for a cheat to maintain stealth is to have the game run virtualized and to implement the cheat in the virtual machine code or hypervisor. With hardware support for virtualization, such an approach can make detection extremely difficult [13].

Cheat Engine and Hoglund's Implant are two examples of cheating systems that use hardware-based methods.

4. MEASUREMENTS

In order to tackle the problem of cheating, our approach is to enable randomly-timed measurements of the target game process that will eventually reveal the presence of individual cheats. The key difference of this hardware measurement approach from current approaches is that these checks are more difficult to circumvent than if the checks were done by a software based cheat detection system. This is because the hardware entity performing the checks is tamper-resistant, and cannot be disabled or circumvented by the platform using software-based attacks.

4.1 CPU Registers and Execution Behavior

One of the key measurements that could be used to detect cheating methods is to measure the CPU state and its execution history. While current implementations of Intel's ME preclude the direct measurement of CPU state, it is useful to examine what could be measured if such a facility were placed within the processor.

- *Instruction Counts*: By tampering with the execution of the game, cheats inevitably perturb instruction usage. Instruction counts, whether done in aggregate or done per assembly instruction opcode can provide an indirect measure of code integrity. For example, hot patches to running code using detours likely use a far JMP instruction since the new code is often loaded in an area of memory far from the original code. Since such an opcode would not be used in the original game, consistent and persistent use of this form of the jump instruction would indicate tampering. Another example would be the anomalous use of floating point instructions for calculating character movements for bots. By measuring histograms of opcode usage, a skew towards extra floating point operations would indicate cheating. Finally, several cheats tamper with the game by injecting debugger instructions (i.e. INT 1 or INT 3) into the game code in order to trap execution to a debugger. Since a debugger should not be attached to the running game, the use of either instruction would immediately indicate game tampering.

- *EIP Behavior*: By virtue of owning the code, game developers know exactly what code should be executed and how it should be run on the client. One key aspect of execution is the virtual address of each instruction a game client might execute either as part of the original executable or in any of the numerous dynamic link libraries. For IA-32, these instructions are fetched based on the instruction pointer EIP register. By injecting code into the memory space of the game process, cheat software will force instructions to execute from unexpected locations in memory. One way to detect such injection would be to measure and report the ranges that the EIP takes on and to compare it against known valid ranges for the game process. Such a measurement could be sampled randomly or recorded before/after every branch, jump, function call, or return call.

- *System Call Behavior*: Another measurement that could be used to validate proper execution is system call traces. This is especially useful for kernel-based cheats that redefine particular system calls in order to execute injected code. Similar to existing tools such as Sebek and Snare, one could examine system call sequences and frequencies in order to detect particular cheats.

- *Code Timing*: Whether it is attaching through the debugger or injecting code into the event loop of a game client, cheats often perturb the execution time of a game. In many cases, cheats add consistent and persistent delay to particular code paths. Consider a detour that injects code before every execution of a particular function call. By measuring the execution time of that function, it can become evident that additional code is always being executed.

- *Exception/Interrupt Counts*: Some cheats force a game to run injected code within overwritten exception and interrupt handlers. Consider Hoglund's Supervisor cheat for World of Warcraft that uses an implant in the kernel and uses hardware debugger breakpoints to trigger its code. In Intel's IA-32 architecture, hardware breakpoints are set via registers DR0-DR7 and throw a debug exception that is handled by the debug interrupt handler (INT 1). By measuring the quantity and type of exceptions and interrupts being generated, one could detect the persistent and anomalous use of such debugger exceptions in the game process.

- *Register Monitoring*: One way cheats avoid detection is by modifying hardware registers so that they point to injected code, leaving the original code unmodified to foil simple integrity checks. However, measuring changes to registers such as the Interrupt Descriptor Table Register (IDTR), the Control Registers (CR0-CR3), and the Debug Registers (DR0-DR7) provides a means for detecting such cheats.

4.2 Memory

Another class of measurements for detecting cheats is the measurement of memory and I/O state. Such a facility can be provided by the current instantiation of Intel's ME by virtue of its location in the memory controller. Specifically, measurements of interest include:

- *Code Integrity*: Since cheats can modify the original instructions after they have been loaded into memory, random integrity checks on the code that is being executed can reveal tampering. In particular, code in the game's ".text" segment, from loaded DLLs, and in the operating system [12] can be checked for tampering in order to reveal hot patches and detours.

- *Scanning for Injected Code*: Rather than modify the original code, some cheats inject a library into the memory space of a process and then overwrite the value of a function pointer or return address in order to get the game process to execute the injected code. One way to detect this method is to scan the game process' executable memory pages for either known cheat code or for unauthorized code. Such a method is used in Warden, the anti-cheat system that is used by World of Warcraft [8], but is prone to false-positives since it is difficult to differentiate a legitimate library from an illegitimate one.

- *Stack Validation*: Code injection attacks are often implemented by hijacking existing threads or creating new threads in the game process. One measurement that can be used to detect such attacks is to validate the game process' stack. For example, a detour implemented as a far JMP instruction can be detected by examining the current stack frame's return address, finding the argument of the last CALL instruction (to locate the address of the function the game process should be in), and comparing against the current value of the instruction pointer. For a function that has been detoured, the instruction pointer will fall outside the range of the original function. Stack validation can also detect attempts by cheats to hijack the game process via rewriting return addresses on the stack. Similar to examining EIP behavior, given an inventory of the location of the game's code and libraries, any return address outside of valid ranges could be flagged. Finally, stack validation is also useful for cheats that inject new threads of execution that directly call game functions such as those used in Hoglund's Supervisor where the stack trace will form an invalid call path when compared against what should be possible from the original code.

- *Function Pointer Validation*: To pass integrity checks, some cheats avoid modifications to code segments and instead modify commonly used function pointers in order to run. One way to detect such tampering is to selectively validate function pointers in the game process. By examining the addresses contained in these pointers and ensuring that they point to locations containing legitimate game or library code, one can detect function pointer tampering. Such validation is commonly done by anti-virus and anti-cheat systems especially for critical function tables such as Import Address Tables, the Interrupt Descriptor Table, and the System Services Dispatch Table.

- *Static Game Data Validation*: It is often computationally too expensive to validate all of the data a game uses. As a result, many cheats target a game process' data sections. For cheats that modify static game data from valid values to invalid ones, checking the integrity of invariant data at run-time can reveal illegal modification. Specific examples include invariants such as a game's gravity constant or memory mapped textures.

- *Memory Watchpoints*: While validating memory locations for cheats that change game data persistently is effective, there are subtle forms of memory tampering that require a more powerful means of detection. Information exposure cheats that do not modify memory contents and team switching cheats that dynamically toggle a memory location between valid values are extremely difficult to catch. Memory watchpoints are one way of detecting such subtle forms of tampering. Although they don't currently exist in the Intel ME, memory watchpoints could capture the time, source, and frequency of access to specific memory locations throughout a game process' lifetime. For example, a watchpoint could log the address of each instruction (i.e. the value of the EIP) that reads or writes from a specific memory location. This would reveal some information exposure cheats where data is read from memory from unexpected locations such as via the Windows `ReadProcessMemory()` API. In addition, a memory watchpoint that tracked the number of changes to a particular memory location or page would be useful in detecting temporary modifications to code pages, cheats that toggle data locations between valid values, or the number of times a debugger has attached to a process. Note that one potential obstacle that must be overcome with this technique is the use of registers and caches for temporary storage of data.

4.3 File and I/O System

Measuring the file system and peripherals can also detect certain types of cheats. Measurements could include:

- *File Integrity*: Ensuring the integrity of the game binary, game data, common libraries, and operating system via cryptographic checksums can detect any cheats that involve replacing original game or system files with those that implement the cheat.

- *Registry/Environment Validation*: Cheats can modify system registry settings or environment variables associated with the game. For example, by changing library search paths, a cheat can cause its libraries to be loaded before the legitimate game libraries. Scanning registry settings that affect the game can detect such cheats.

- *I/O Path Validation*: As shown previously [10], one way to detect automation cheats is to compare the raw I/O signals coming from the hardware to the actual input being given to the game engine. Another example for detecting wall-hacks is to take remote screenshots of a game client to visually inspect what is being displayed by the game client [2].

Cheat	Detection Method
Authorized Data Read	Memory Watch Points
Unauthorized Data Read	Memory Watch Points
Unauthorized Data Write	Static Game Data Validation Memory Watch Points
Code Injection	Stack Validation Code Integrity Scanning for Injected Code Registry Validation Instruction Counts Code Timing
Thread Manipulation	Stack Validation Code Integrity EIP Behavior
Direct Function Call	Stack Validation EIP Behavior System Call Behavior I/O Path Validation
Function Pointer Hook	Function Pointer Validation Stack Validation EIP Behavior Memory Watch Points
File Modification	File Integrity Code Integrity
Hardware Facilities	Register Monitoring Exception/Interrupt Counts

Table 1: Cheats and the methods that detect them.

4.4 External Processes

Another useful measurement would be to look outside of the game and inspect external files and processes that potentially affect the game process. This method is difficult and dangerous. Several anti-cheat systems such as Warden [4] and Punkbuster [2] inspect the file system and the memory image of *all* other running processes for known cheats. This has dire consequences regarding personal privacy if sensitive information is read and leaked. Another problem is the danger of false positives. Since it is infeasible for such a method to know the structure of every possible application running on a client machine, the technique is left to scan for signatures that identify cheat code. In one case involving Punkbuster, a devious hacker issued a broadcast IRC message to a large number of players containing a binary pattern matching a cheat signature that Punkbuster was checking. As a result, more than 300 members of the IRC channel were incorrectly banned [9].

5. THREAT ANALYSIS

The usefulness of the hardware measurement based approach can be assessed by examining how effective it is at detecting the cheating methods currently in use and comparing this to the cost faced by cheaters seeking to circumvent the approach. The un/authorized read methods for cheating could potentially be captured with memory watchpoints. Some forms of unauthorized write attacks could be detected by simple integrity checks, however, subtle forms that do not change memory locations to invalid values remain undetected unless finer-grained memory watchpoints are supported. Direct function call cheats are detected by stack validation while the code injection attacks are detected using measurement methods for stack validation and code integrity. Hooks on function pointers that have known "good" values are detected using integrity checks of pointers in memory and the code they point to. File modification attacks are addressed by the I/O system protections while cheats exercising hardware facilities could be detected if the hardware supported a register monitoring capability. Table 1 summarizes the cheat methods and the potential hardware measurements that can be used to detect them.

6. CONCLUSION

This paper has examined an approach for detecting cheats using stealth hardware-based measurements. Towards this end, we have examined a range of methods that hackers use to compromise games as well as a range of measurements that hardware might support in order to detect these methods in a resilient manner. Cheaters willing to expend resources to wage a hardware modification attack could successfully circumvent the system, however the system detects cheats in a manner that would be much more expensive to circumvent than existing solutions due to the tamper-resistant nature of the measurement hardware.

7. REFERENCES

[1] Cheat Engine Developers. Cheat Engine. `http://cheatengine.org`.

[2] Even Balance, Inc. PunkBuster Online Countermeasures. `http://evenbalance.com`.

[3] J. Evers. Taking on Rootkits with Hardware, December 2005. `http://news.cnet.com/Taking-on-rootkits-with-hardware/2008-1029_3-5992309.html`.

[4] G. Hoglund. Keeping Blizzard Honest - Announcing the Release of 'The Governor', 2005. `http://www.rootkit.com/`.

[5] G. Hoglund. Hacking World of Warcraft: An Exercise in Advanced Rootkit Design. In *Black Hat*, 2006.

[6] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *USENIX Windows NT Symposium*, July 1999.

[7] Intel. Intel Active Management Technology. `http://www.intel.com/technology/platform-technology/intel-amt/`.

[8] Lavish Software. On Warden Blog. `http://onwarden.blogspot.com/`.

[9] netCoders. The Unerring Punkbuster... `http://forum.netcoders.cc/announcements/14061-unerring-punkbuster.html`.

[10] T. Schluessler, E. Johnson, and S. Goglin. Is a Bot at the Controls - Detecting Input Data Attacks. In *NetGames*, October 2007.

[11] Solar Designer. Getting Around Non-executable Stack (and Fix), August 1997. Bugtraq Mailing List.

[12] T. Schluessler et. al. Runtime Integrity and Presence Verification for Software Agents. In *Technology@Intel Magazine*, December 2005.

[13] D. Zovi. Hardware Virtualization-Based Rootkits. In *Black Hat*, 2006.