Fides: Remote Anomaly-Based Cheat Detection Using Client Emulation[*]

Edward Kaiser
Portland State University
Portland, OR, USA
edkaiser@cs.pdx.edu

Wu-chang Feng
Portland State University
Portland, OR, USA
wuchang@cs.pdx.edu

Travis Schluessler
Intel Corporation
Hillsboro, OR, USA
travis.schluessler@intel.com

ABSTRACT

As a result of physically owning the client machine, cheaters in online games currently have the upper-hand when it comes to avoiding detection. To address this problem and turn the table on cheaters, this paper presents Fides, an anomaly-based cheat detection approach that remotely validates game execution. With Fides, a server-side Controller specifies how and when a client-side Auditor measures the game. To accurately validate measurements, the Controller partially emulates the client and collaborates with the server. This paper examines a range of cheat methods and initial measurements that counter them, showing that a Fides prototype is able to efficiently detect several existing cheats, including one state-of-the-art cheat that is advertised as "undetectable".

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive Software

General Terms

Security

Keywords

Online Games, Anti-cheating, Anomaly-based Detection, Remote Measurement, Partial Client Emulation

1. INTRODUCTION

1.1 The Cheating Problem

Multiplayer online games simulate complex environments. Due to limited server computation and player sensitivity to network latency, games are designed to offload computation to the client which is expected to run accurately and keep secret state hidden from the player. Cheat software violates this trust by altering the local simulation to give the cheater an unfair advantage.

[*]This material is based upon work supported by Intel Corporation under Grant 34557 and the National Science Foundation under Grant CNS-0627752. Any findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Intel or the National Science Foundation.

While the motives vary, cheating has become widespread. Many cheat communities write and sell cheats which range from automated bots that treasure hunt virtual items, level up characters, and attain ranks for cheaters unwilling to play the game, to cheats that provide perfect aim, reveal secret knowledge, and force opponents to disconnect (and thereby concede) for cheaters unable to win unaided. Legitimate players are frustrated by cheaters to the point where they seek other games that are more resilient to cheating. Since losing existing and potential paying players directly impacts revenue, developers must thwart cheating in their games.

1.2 A Distinct Security Problem

Cheaters have one clear advantage over developers – they control the machines on which they cheat. This means cheat software is granted all the necessary privileges and may run before anti-cheat software is ever loaded. State-of-the-art cheats conceal their presence by modifying the operating system, disabling or spoofing anti-cheat software, and even cloaking their code just before routine anti-cheat software runs. Fortunately, *cheats are a weak threat compared to other security problems* like rootkits, botnets, and worms.

At the granularity of individual cheaters, the problem is not urgent. There is no need for rigorous cheat prevention or immediate containment; sensitive information (such as passwords and personal information) is not being stolen and the machine is not being used to attack other network hosts. To easily access game data and functionality, cheats embed themselves inside the game process, drastically limiting their ability to conceal their presence indefinitely.

Cheating damage is easily undone; by confiscating their ill-gotten gains and disabling their account, a cheater can no longer affect legitimate gameplay. The full extent of cheating damage is easily determined and cleanup is comprehensive. Unusual for a network adversary, cleanup directly punishes the cheater, since disabling their account annuls all invested time and confiscates both the software purchase and paid subscription fees.

Due to the long-term connected nature of online games, the server has many lengthy client interactions where the cheater need only be detected once to halt their disruption. Reversing an adversary's traditional advantage, *the cheater must anticipate and guard against every detection technique to succeed while the developer need only detect a single unauthorized change to thwart them.* With no need for urgency, being able to eventually detect the cheater is sufficient to address the cheating problem, which is supported by the widespread use of cheat detection.

1.3 Contributions

This paper presents Fides[1] which is, to our knowledge, *the first anomaly-based cheat detection approach* in literature and useable on commercial-off-the-shelf online games. The system is designed to dovetail with prevalent client-server game architectures in order to minimize the modifications necessary for adoption and efficiently detect cheats that directly alter or automate game clients. Cheating external to the game software (e.g., cheating through collusion or "roboting") is not addressed by this approach.

Fides is a promising generalized cheat detection approach that is game-independent; the approach works across games, game genres, operating systems, and hardware architectures. Fides applies anomaly-based application integrity research to the game domain avoiding the detractions of existing signature-based detectors. Specifically, Fides does not require human intensive maintenance and can quickly detect cheats without first knowing their operational minutia.

Anomaly-Based Cheat Detection. Similar to other security problems, there are two general cheat detection methodologies. *Signature-based* detection learns what cheats look like and actively searches for those patterns while the game is running – scanning not only the game but every active process on the system. Anti-spyware tools observe that this maintenance-heavy methodology is the current choice of game developers [13], likely because it can be implemented after game release.

In contrast, *anomaly-based* detection understands how the legitimate unmodified game operates and periodically audits the client searching for unexpected deviations that indicate the presence of cheat software. This detection methodology only inspects the client process which is feasible since most cheats currently embed themselves in the client process. As a result the search space is well bounded; other processes do not affect the speed nor accuracy of this methodology.

Restricting the search space to the game process yields security benefits too. Primarily there is *no risk of privacy breach* [20]; sensitive data that exists in foreign processes is not read and cannot be leaked. Additionally, adversaries cannot exploit the scanning of unfamiliar processes namely by injecting false positives. This avoids attacks similar to one where an IRC message containing the binary pattern of a cheat signature was broadcast to a clan channel, was falsely detected by Punkbuster, and resulted in more than 300 legitimate players being incorrectly banned [22].

Anomaly-based detection is *cheat agnostic* and requires fewer developer resources (i.e., manpower and storage) to maintain as the game is readily accessible for study by its own developers – cheats do not need to be captured and studied to create signatures as only knowledge about the game is used. The perpetually increasing collection of cheats, cheat variations, and polymorphic cheats does not inflate the knowledgebase which only changes when the game does.

Furthermore, anomaly-based detection is *not reactionary.* Signature-based detection relies on manually finding cheats and cataloging their signatures causing a lag between when a cheat is first used and when its signature can be detected. This gap is often artificially increased by the developer to avoid tipping off cheaters as to when and how they were caught, however, legitimate players may perceive that personally observed cheating behavior is going unpunished.

[1]Fides was the goddess of trust in Roman mythology.

For anomaly-based detection to work properly, all host variation that affects the game client (i.e., library versions and mapped locations) must be accommodated so that all legitimate game operation is recognized and will not be misclassified as cheating. With such accommodations, anomaly-based detection may be advantageous in terms of efficiency, maintenance costs, and accuracy.

Continued Random Remote Measurement. A novel feature of Fides is that it performs continued random remote measurements of the game client during gameplay. Towards this approach, the system comprises a simple client-side Auditor that is directed by a robust server-side Controller. The Auditor supports a selection of parameterized functions to measure the client and return the results to the Controller. The system complexity is located in the Controller which dictates the audit strategy (i.e., what gets measured as well as when), and validates all measurements taken.

Minimizing the Auditor's complexity allows the developer to strengthen it against attack through the use of execution entanglement, rapid polymorphism, and lightweight tamper-resistant co-processors. These techniques, discussed further in Section 5.2, can provide better assurances about the Auditor's integrity and measurement accuracy.

Additionally, employing a partially randomized strategy and placing it in the Controller avoids telegraphing audits and allows the developer to change strategy surreptitiously. The fear of the unknown, specifically not knowing when cheat techniques believed to be detection-proof become obsolete (with dire consequences for the cheater), has been shown to be an effective deterrent for would-be cheaters [14]. Continued random measurements will eventually detect a persistent cheater and such a strategy is suitable due to the always-on nature of the game client.

Partial Client Emulation. Another novel feature of the Fides system is that the Controller partially emulates the game client to accommodate any client system variation that affects measurement validation. The Controller includes routines for mapping the client's virtual memory and learning its execution patterns during player logon. This compiled knowledge is relevant for the duration of gameplay and is used to validate each Auditor measurement; deviation indicates the presence of cheat software.

Static game data and code are learned by parsing binary and library files, and rebasing them to match the client which must be designed to relate pertinent library details at logon. Dynamic data is semantically identified using source code and debugger files so that the Controller may query the server to corroborate those values, meaning the server must be designed to respond to such queries.

Client execution patterns are learned by disassembling the mapped code sections and creating a graph of legitimate execution describing the range of and relationship between all client functions. This knowledge allows the Controller to know what code should be executing given an instruction pointer, and whether specific code locations represent `CALL` instructions linking two legitimate functions.

To better learn commercial-off-the-shelf games (which may be obfuscated and for which one may not have source files) that run in common environments, the emulator includes tools for sampling and profiling legitimate client execution in a secure server-side environment. These tools can provide client understanding where static learning routines cannot.

2. CHEAT METHODS

This section reviews formerly surveyed cheat methods [9]. While this paper references Windows specific functions and Intel® specific architecture features, these methods apply to other operating systems and architectures.

2.1 Altering Memory Contents

Code Manipulation. Many cheats change the game functionality by overwriting (i.e., *hot patching*) existing code, usually via the `WriteProcessMemory()` function. Often hot patches are used to redirect execution to more complex cheat code, however, some cheats can be implemented in place and do not require additional code. For example, cheats that disable undesirable game mechanics (such as flash grenade blinding effects) merely replace select game instructions with NOPs. The instructions that call the undesired function are typically overwritten rather than the function itself.

Code Injection. Cheats that involve relatively small changes but cannot be done in place usually inject code into pockets of allocated but unused executable memory (i.e., *code caves*). This only works if there is enough allocated but unused memory. More substantial cheats must allocate new pages via `VirtualAllocEx()` and make them executable via `VirtualProtectEx()` before code can be injected with `WriteProcessMemory()`.

DLL Injection. One method for injecting substantial amounts of cheat code into the game process is Dynamic Link Library (DLL) injection [10, 31]. This method is popular due to the ease of writing and loading libraries. There are several ways that a library containing cheat payload can be loaded; calling the `LoadLibrary()` function, hooking `LoadLibrary()` as it is used to load legitimate game DLLs, or modifying the `AppInit_DLL` registry entry to automatically load the library at game start up.

Data Manipulation. Some cheats directly modify game data using functions such as `WriteProcessMemory()` [3]. Often these cheats target *static data* (e.g., the gravity constant to enable wall-walking, or memory-mapped wall textures to make them transparent) even though static data modification is easy to detect since modified values are made invalid. Other cheats toggle *dynamic data* between two or more valid values in an illegitimate way (e.g., quickly alternating between teams to trick the game client into revealing enemy locations via the radar).

File Replacement. A historic method (surprisingly still employed) involves modification of files on disk before they are loaded into memory, such as the game binary and data files, or libraries which the game uses. For example, some wall hacks replace texture files with transparent alternatives.

2.2 Altering The Execution Path

Thread Injection. Once cheat code is injected into the game it must be executed. Cheats that operate in parallel with game code (e.g., automation cheats) accomplish this by injecting a new thread into the game process using the system function `CreateRemoteThread()`.

Thread Hijacking. More commonly, an existing thread is temporarily hijacked using a *detour* (or *trampoline*) [15]. This involves redirecting execution by hot patching a few bytes of a game function to jump to cheat code. After the cheat has finished, execution returns to the original function.

Function Pointer Hooking. A thread may alternatively be hijacked by *hooking* a function pointer; the pointer used to call a game function is redirected to instead point to cheat code. Susceptible function pointers are prevalent in any running process and include:

- Return addresses on the stack. Modification returns execution to injected code rather than to the original caller. This is the basis for return-to-libc attacks [32].
- Function pointers that implement run-time binding of operations or jump table implementations of switch statements in compiled C and C++ code.
- Import Address Table (IAT) entries that point to the specific functions imported when loading linked DLLs.
- Interrupt Descriptor Table (IDT) entries that point to interrupt handlers.
- System Service Dispatch Table (SSDT) entries that point to system call functions.
- I/O Request Packet (IRP) Function Table entries that point to the kernel's I/O processing functions.
- Structured Exception Handler (SEH) values on the stack that point to exception handlers.
- Windows' inter-window messaging function pointers. These are easily hooked using the `SetWindowsHookEx()` function.

Direct Function Calls. Rather than re-implement game or system functions, cheat code often directly calls those functions. This is especially true for automation cheats which collect game state, make game-play decisions, and then call game functions to take action.

Software Debugging. Some cheats attach to the game process as a debugger using `DebugActiveProcess()`. The game can then be suspended by injecting `INT3` breakpoint instructions and waiting for debugger breakpoint exceptions with `WaitForDebugEvent()`. At that point, execution can be trivially redirected via `SetThreadContext()`.

2.3 Hardware Facilities

Hardware Debugging. Modern processors support debug registers that stop execution and throw a breakpoint exception whenever the specified code locations are reached or the specified memory locations are accessed. Cheats (like Hoglund's World of Warcraft "Supervisor" [14]) use this facility to hijack execution without injecting any breakpoint instructions into the game code.

Memory Management Manipulation. To gain access to memory, cheats may manipulate the processor's memory management subsystem including the control and segment registers. For example, tampering with the IA-32 control registers (`CR0` - `CR3`) allows cheats to modify pages flagged as read-only.

Register Manipulation. Some cheats exploit hardware registers to avoid hooking function pointers. For example, by modifying the Interrupt Descriptor Table Register (`IDTR`) which points to where the IDT resides, a cheater redirects interrupts to a completely different table containing pointers to cheat code while leaving the original IDT unaltered.

3. THE FIDES SYSTEM

This section presents Fides, a novel system for detecting cheaters in online games. The system performs anomaly-based detection via remote measurements that are validated through client emulation and server corroboration.

3.1 Architecture Overview

Fides works by auditing the client and validating those measurements against a set of expected legitimate values. Figure 1 shows the system's two components: the client-side *Auditor* takes measurements of the game client when the server-side *Controller* requests them. The Controller decides what to measure and when, validates measurements through client emulation and server corroboration, and logs cheat activity to the cheater's account. The communication channel between the Auditor and Controller must be authenticated and encrypted so that detections may not be attested as the results of possible man-in-the-middle attacks.

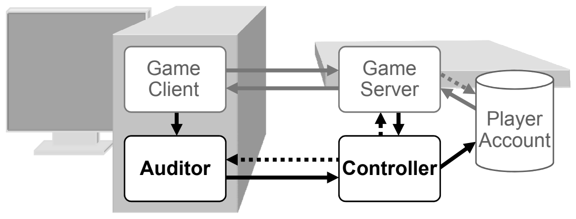


Figure 1: Game interaction after the addition of the Auditor and Controller. Dashed arrows represent request traffic, while solid ones represent data flow.

While the Auditor and Controller are shown as separate from the game, they could be implemented within the client and server. Locating the Auditor in the client enables the polymorphic patching discussed in Section 5.2. Locating the Controller in the server streamlines the validation of dynamic data.

3.2 The Auditor

The Auditor (shown in Figure 2) accepts the Controller's instructions, performs the requested measurements, and returns the results. Table 1 highlights the cheat methods best detected by each of the initial measurement routines.

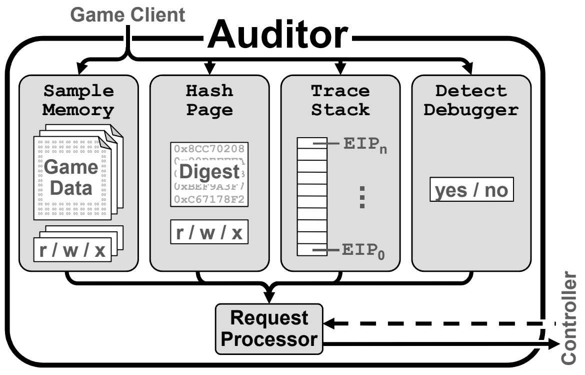


Figure 2: The Auditor and its initial measurement routines for collecting client state.

Measurement	Cheat
Sample Memory	Dynamic Data Manipulation
Hash Page	Static Data Manipulation Code Manipulation Code & DLL Injection File Replacement Memory Management Manipulation
Trace Stack	Thread Injection Thread Hijacking Function Pointer Hooking Direct Function Calls Register Manipulation
Detect Debugger	Software Debugging Hardware Debugging

Table 1: Summary of cheat methods best detected by the initial measurements of the Auditor.

3.2.1 Auditor Measurements

Sample Memory. This routine returns the contents and read-write-execute permission flags of the specified memory range, facilitating the detection of cheats targeting dynamic data. Since the range contains dynamic data, every client thread must be briefly suspended to get a quiescent reading.

Hash Page. This routine uses a cryptographic hash function (e.g., SHA1 [21]) to hash the specified memory page and return the digest along with the page's permission flags, facilitating efficient detection of cheats targeting static data and code. If unspecified, the page currently executed by a randomly selected thread is hashed. This measurement does not require suspending any client threads because the target memory page should be static.

Trace Stack. This routine suspends one client thread, chosen at random if unspecified, and obtains the current instruction pointer (`EIP`) and stack pointer (`EBP`) which are the representative registers of the stack frame (i.e., client function executed). Recording all encountered instruction pointers, the routine recursively descends the stack, obtaining each previous stack frame by dereferencing the stack pointer of the current frame, until it reaches the frame corresponding to the thread entry point (i.e., `EBP` = NULL) at which point the measurement is done and the client thread may be resumed. The `n` recorded instruction pointers list the function calls that the client thread used to get from the entry point to the current point of execution and are returned to the Controller, facilitating the detection of cheats that hijack game execution, redirecting it to unrecognized locations or between unrelated functions.

Detect Debugger. This routine detects whether the game is being manipulated by a cheat attached to it as a debugger, and returns the Boolean result to the Controller. If the game has a debugger attached, the corresponding flag in its Process Execution Block (PEB) should be set to true (though most cheats set it to false) which can be tested via the `CheckRemoteDebuggerPresent()` function. The routine may attach to the client (even if located within the client) using `DebugActiveProcess()` which always fails if another debugger is present. Attaching as debugger is relatively expensive (roughly $9ms$) and should only be done if the first test is negative. This measurement detects all software debugger cheats and most hardware debugger cheats.

3.3 The Controller

The Controller (shown in Figure 3) hosts the complexity of the Fides system. Its *Client Emulator* learns the game client properties that remain constant (i.e., static data, code, and function relationships) and identifies dynamic data sections. The *Audit Strategist* uses the compiled knowledge to orchestrate a game-specific audit strategy during gameplay. The *Audit Validator* uses emulated state to verify audits of static client properties and corroborates with the server to verify audits of dynamic client state.

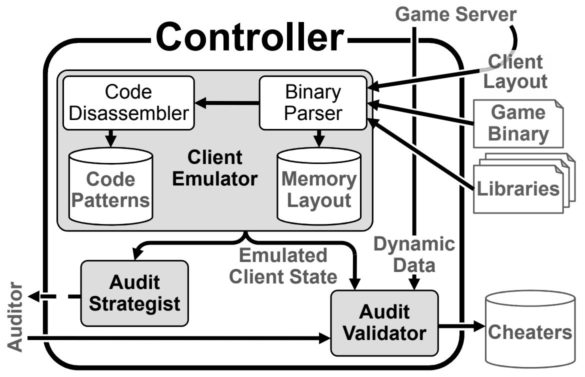


Figure 3: The internal operation of the Controller. Arrows show the data flow between external data sources and the Client Emulator, Audit Strategist, and Audit Validator components.

3.3.1 Client Emulator

The hardware and software of the machine on which the client runs varies from player to player, and even changes between play sessions for individuals who play on different machines. Such variation affects the memory layout and contents of the client process. For example, the libraries loaded by the game differ between operating systems and system versions. Additionally, memory layout differs between executions on systems that randomly rebase libraries via Address Space Layout Randomization (ASLR) [30].

The Client Emulator accommodates for host variation by thoroughly mapping the code and static data and identifying the dynamic data for each client at logon. Clients must communicate the pertinent layout details (i.e., the name, version, and base location of every library loaded) to the Controller via the server. The emulator owns a copy of every known legitimate library so the client need not relay actual files. The client is behooved to accurately report its layout otherwise inconsistencies will be detected as cheating.

Game clients may run on operating systems without ASLR (or with ASLR disabled) and share common library content and layout. Core system libraries (e.g., `KERNEL32.DLL` and `NTDLL.DLL`) often contain thousands of small functions that comprise much of any client application's complexity. The emulator can leverage commonality to reuse significant portions of the emulated state between clients, and learn only truly variable client structure at logon.

The Client Emulator uses two routines for learning client structure: a *Binary Parser* and a *Code Disassembler*. The emulator also contains an *Execution Sampler* and *Execution Profiler* to better understand the client application when it runs in a cheat-free environment.

Binary Parser. The primary learning routine is the Binary Parser which maps the client virtual memory from the executable, all linked libraries, and data files. Using the client-provided layout information, the routine rebases the libraries to know the location and properties (i.e., read-write-execute permissions) of every memory section in the client. Function pointers (e.g., IAT pointers) are corrected so that page hashes may be generated for static data and code sections, reducing emulator state for those pages to a small digest later used to validate page hash audits. The emulator uses source code and debugger database files to learn the semantics of variables in dynamic data sections. This information helps the Controller properly query the server about the correct client state during the validation of memory sample audits. Clearly, the game server must be designed to corroborate queries about dynamic variables.

Code Disassembler. The secondary learning routine is the Code Disassembler which uses the code sections mapped by the Binary Parser to learn the range of and relationship between every function in the game client. The routine starts at the entry point of the executable and traverses the code, observing the memory range of the each function and the location of instructions that relate functions, creating an execution graph (like the one shown in Figure 4) similar to the "callgraph" model [34]. This knowledge is used when validating stack trace audits.

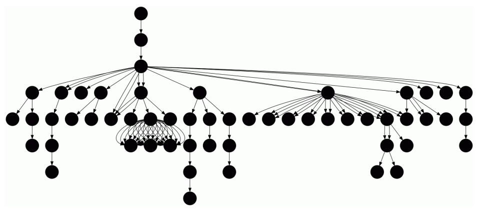


Figure 4: One game's partial execution graph. Nodes represent functions while edges represent the `CALL` instructions that relate any two functions.

Execution Sampler. The above routines are sufficient to learn client applications for games designed to work well with Fides, however, to better learn commercial-off-the-shelf games (which may be obfuscated and for which one may not have source code) running in non-ASLR environments the emulator includes an Execution Sampler. The tool executes the client application in a cheat-free environment, like that which the actual client should be running in. The sampler then exhaustively hashes all the non-writeable pages of the game, furnishing the knowledge to the emulator so it may be used to validate page hash audits of the actual client.

Execution Profiler. The emulator also includes an Execution Profiler to learn the instruction range of and the relationship between client functions (specifically observing indirect function calls), reinforcing the execution graph used to validate stack trace audits. This tool executes the client application in a secure cheat-free environment, attaches to it as a debugger, and uses hardware debugging routines (i.e., register manipulation and single-step interrupts) to step through its execution. The profiler records details about instruction counts, code timing, and function execution frequencies. These execution patterns may also be used to improve audit strategies.

3.3.2 *Audit Strategist*

The Audit Strategist orchestrates the detection strategy, dictating the ordering, timing, and details of every audit request. Locating all the strategy in the Controller avoids telegraphing measurements, allows the developer to change strategy surreptitiously, and minimizes Auditor complexity. The strategy may be game-specific using developer intuition regarding data or code likely to be attacked, however, a good strategy must employ some randomization to prevent cheaters from predicting audits and developing a successful cloaking routine. As cheaters will be audited countless times while they are online, a strategy with randomness will, with good probability, eventually catch them red-handed.

3.3.3 *Audit Validator*

After measurements have been returned to the Controller, the Audit Validator validates them through different means according to their type as follows.

HASH PAGE. These audits measure static data, so they should remain unchanged. The validator does a simple binary comparison of the audited digest against the stored digest. Any bitwise difference means that the page has been modified, indicating cheat behavior.

SAMPLE MEMORY. Although an easy audit to perform, validating sampled dynamic data is difficult and involves corroboration by the server. The validator must determine the semantics of the data (i.e., which variable or structure it represents) and query the server regarding what the proper value or range of values should be.

Consider for example, the cheat that frequently toggles the team variable to reveal enemy locations via the radar. To validate an audit of that variable, the validator must request from the server which team the player is supposed to be on and compare the two values. This is a straightforward example because there only two valid options and the variable should change very infrequently, perhaps on the order of once per match.

Next consider, teleport cheats that adjust the player's in-game coordinates. Validating an audit of this data must request from the server what it believes the proper value should be. However, in this case there are innumerable valid values, the value changes frequently, and due to dead-reckoning techniques a legitimate client will have a value a few steps ahead of that held by the server. In this case, the server must provide a range of locations where the player could legitimately be according to game physics.

STACK TRACE. Validating a stack audit involves walking the execution graph according to the sequential list of instruction pointers returned by the audit. If an instruction pointer calls an unknown location in memory (referencing a nonexistent graph node) or represents a call to a function from an unrecognized location (referencing a nonexistent edge from the last visited node) client execution has been high jacked. Indirect function calls are handled by validating every direct call up to it and immediately (without resuming the client thread) hash auditing the functions called after it ensuring that they have not been detoured.

DETECT DEBUGGER. The simplest audit to validate; a detect debugger audit should be false otherwise the client is being manipulated by a debugger.

4. EVALUATION

4.1 Software Prototype

To demonstrate the utility and easy deployment of the Fides approach, a software prototype was implemented consisting of two applications (corresponding to the Auditor and Controller) that communicate through a TCP socket. The prototype is game-agnostic and has been tested successfully with several games.

AUDITOR DETAILS. The Auditor employs functions of the Windows debugging API to conveniently read client virtual memory using `ReadProcessMemory()` and access the client's registers using `GetThreadContext()`. When necessary, the Auditor suspends a client thread using `SuspendThread()` which is three orders of magntitude faster than attaching as a debugger.

CONTROLLER DETAILS. The Binary Parser processes files of the Windows Portable Executable (PE) format, maps those files to virtual addresses, fixes the IAT for each file, and hashes the static sections with SHA1. The implementation challenges included resolving circular linking dependencies, evaluating forwarded exports, handling aliased functions, and locating anonymous import tables. When available, the parser also processes files of the Program Database (PDB) format to uncover un-exported function symbols and identify dynamic data variables.

The Code Disassembler is a basic `x86` disassembler which walks client code starting at the entry point of the binary and following function calls to determine the range of and relationship between client functions. Implementation challenges included resolving functions that terminate in inconsistent ways, functions that use discontinuous memory, functions that embed return instructions in the middle (in particular those protected by the Windows implementation of StackGuard [5]), and handling IAT function calls that vary between 2 levels of indirection and 3 levels of indirection.

To better understand the client structure, the Execution Sampler uses `CreateProcess()` to run the client application local to the Controller – assuming for portions of this evaluation that the client and Controller share a common non-ASLR operating system (i.e., Windows XP SP3). The sampler exhaustively scans for and hashes every user-space page (between `0x00000000` and `0x7FFFF000`) that is static – allocated and non-writeable.

The Execution Profiler also runs the client, but attaches to it as a hardware debugger to learn the function frequency and indirect function calls unhandled by the disassembler. The implementation challenges included avoiding the game's anti-debugging techniques, resolving timing delay caused by the debugger interrupts, and guiding the game into its gameplay loop to be measured.

MEASUREMENT STRATEGY. The Controller uses a straight forward game-independent audit strategy that takes measurements at a specified interval. To avoid predictability, timing jitter is added (uniformly random between -5% and +5% of the interval length). Throughout each experiment the chosen measurement routine was kept constant, but the target of each individual measurement was left unspecified. For a stack trace audit, this simply means a thread was selected at random. For a hash audit, this means a thread was selected at random and its executing code page was hashed.

Auditor Task	Cycles		Time
Null	**82 ±**	**18%**	**34.6ns**
Sample Memory	**87,941 ±**	**21%**	**36.8μs**
Suspend Threads	54,960 ±	22%	22.9μs
Read Memory	11,947 ±	35%	5.0μs
Fetch Page Flags	21,854 ±	30%	9.1μs
Resume Thread	4,249 ±	97%	1.8μs
Hash Page	**112,885 ±**	**21%**	**47.2μs**
Suspend Thread	4,843 ±	33%	2.0μs
Get EIP	19,760 ±	44%	8.3μs
Resume Thread	4,153 ±	65%	1.7μs
Read Page	15,597 ±	32%	6.5μs
Hash Contents	55,391 ±	23%	23.2μs
Fetch Page Flags	11,031 ±	42%	4.6μs
Trace Stack	**64,399 ±**	**27%**	**26.9μs**
Suspend Thread	4,844 ±	25%	2.0μs
Get EIP & EBP	26,042 ±	45%	10.9μs
Get Stack Range	15,782 ±	42%	6.6μs
Traverse Stack	13,462 ±	37%	5.6μs
Resume Thread	4,292 ±	31%	1.8μs
Detect Debugger	**23,246,056 ±**	**1.5%**	**9.7ms**
Test Debug Flag	2,998 ±	43%	1.3μs
Attach Debugger	21,411,428 ±	1.2%	8.9ms
Detach Debugger	1,828,041 ±	7.0%	0.8ms

Controller Task	Cycles		Time
Null	**85 ±**	**19%**	**35.6ns**
Parse Binaries	**236,322,896 ±**	**0.1%**	**98.8ms**
(single file)	17,221,136 ±	2.5%	7.2ms
Memory Map File	741,448 ±	5.0%	0.3ms
Identify Sections	4,774 ±	20%	2.0μs
Allocate IAT	388,067 ±	19%	0.2ms
Populate IAT	5,193,318 ±	2.1%	2.2ms
(single entry)	58,517 ±	4.4%	24.5μs
Hash Sections	10,893,211 ±	4.5%	4.6ms
(single page)	46,908 ±	6.7%	19.6μs
Disassemble Code	**205,290,560 ±**	**0.2%**	**85.9ms**
Isolate Function	1,165,818 ±	6.3%	0.5ms
Validate Result	*varies as follows ...*		
Sample Memory	*overwhelmingly data dependent*		
Hash Page	3,170 ±	20%	1.3μs
Trace Stack	10,808,131 ±	10%	4.5ms
Detect Debugger	130 ±	21%	52.4ns

Table 2: Average number of cycles (also expressed as time) to perform Auditor and Controller tasks.

4.2 Experiments and Results

4.2.1 Benchmarking

The routines of the Auditor and Controller were benchmarked to demonstrate that Fides can be incorporated into commercial products without imposing a significant negative compute burden in exchange for its cheat detection benefits. Using a 2.39GHz Intel® Core2 machine, each routine was executed 1,000,000 times recording the average and variance in the number of cycles to complete them, shown in Table 2.

The results demonstrate that the Auditor's measurements complete very quickly. The three most common routines (sample memory, hash page, and trace stack) operate on the order of tens of microseconds, adding imperceptible overhead to the game. The most expensive routine (detect debugger) takes 9.7ms and only adds a perceptible hiccup to the game if done in a tight loop.

The Controller's learning routines take under 100ms and do not impose a meaningful burden to the game considering they are only run once at client logon, and represent the time to learn the real game described in the next subsection. The number of subtasks to learn a game depends on its complexity. This game imports 17 DLLs representing 4103 total functions, many of which belong to system libraries and are not linked. The memory sections represent 1649 total pages of which 1619 are flagged as static and can be expressed by a mere 32KB of hash digests. This indicates library layout commonality (specifically on non-ASLR systems) could reduce per-client emulator state by upwards of 98%.

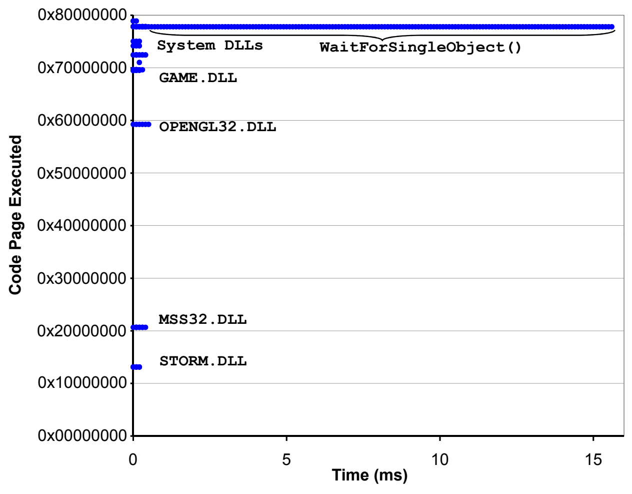
The validation routines are very quick. Validating a hash page audit involves a binary comparison of the 20B digest and validating a debugger audit evaluates the Boolean value, both very efficient routines. Validating a stack trace audit is more involved because each instruction pointer in the list must be looked up as to which function it belongs to, and if it can legally call the next function. The effort to validate such an audit depends on the stack depth and the number of functions in the game.

4.2.2 Commercial-Off-The-Shelf Game: Warcraft III

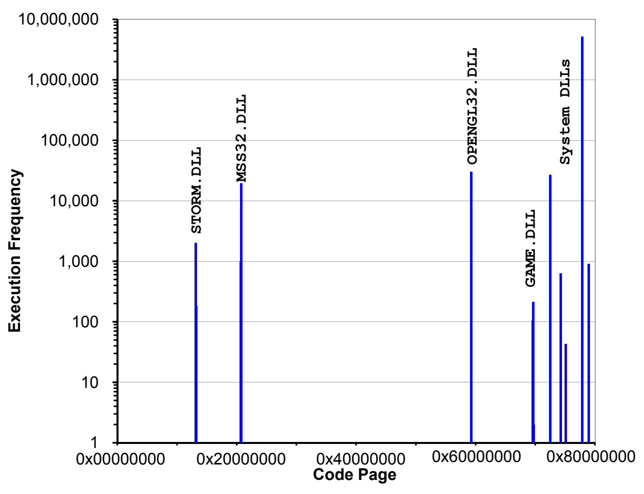
The Fides approach works best on games designed with it in mind. However, to demonstrate that the Fides approach works on commercially deployed games, this section explores the detection of cheats for the popular Real Time Strategy (RTS) game Warcraft III [2]. The game was released in 2002 and has remained extremely popular since then, especially as the game of choice for professional video game competitions. Weekly, thousands of players continue to play it online in competitive ladder-ranked matches. The developer, Blizzard Entertainment, has patched the game regularly throughout the years (the latest patch was in October 2008) to fine-tune game balance and attempt to stay ahead of cheaters.

Warcraft III has several security features that make it complex. The game client is designed so that the executable is simply a launcher, and all game functionality is located in a library appropriately named `GAME.DLL`. This library is loaded at runtime via the `LoadLibrary()` function, and is obfuscated to hinder disassembly. The client is heavily multi-threaded, employing 22 threads of which only 8 are active during normal gameplay. Threads have assigned tasks that include rendering the world, loading map and texture info, gathering player input, managing network connections, running AI for computer opponents, hosting games locally, and performing anti-cheating / anti-debugging (the client attempts to kill any process that attaches as a debugger).

Game Profile. Using the Execution Profiler, the profiles of the threads active during the gameplay loop are shown collectively in Figure 5 (presented on a time axis rather than cycles for readability). On the test system, the game loop takes only 15.6ms to complete, equal to a frame rate of 64.2fps. Game threads execute system functions (in `NTDLL.DLL` and `KERNEL32.DLL`), core game functions (in `GAME.DLL`), render graphics (in `OPENGL32.DLL`), process audio (in `MSS32.DLL`), and access game data (Blizzard games use `STORM.DLL` to load data from disk). The game uses `WaitForSingleObject()` to sleep while waiting for I/O; if an input event occurs the function returns early, otherwise is sleeps for the full duration. Without active input, the game spends roughly 77% of its loop sleeping. With 22 threads to randomly choose from when auditing the game, the probability of hashing game code pages targeted by a cheat is naturally low. An audit strategist aware of these facts could bias against inactive threads to increase the frequency where audits target pages other than the system page containing that idling function.



(a) Time profile of code page execution.



(b) Frequency of code page execution.

Figure 5: Code pages executed during the Warcraft III gameplay loop.

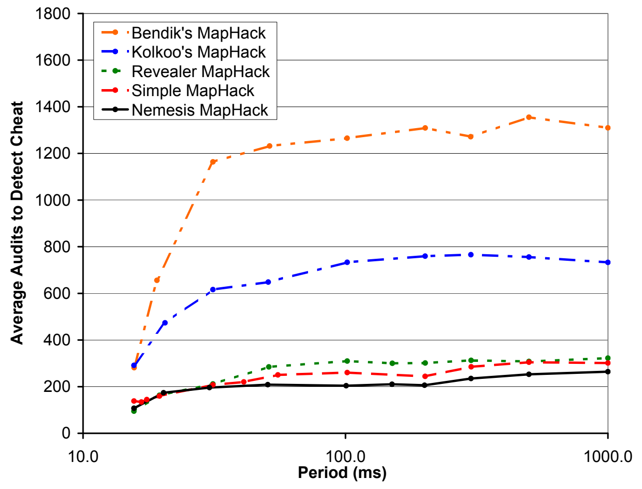


Figure 6: Hash Page detection of select Warcraft III cheats with varying time between audits.

Cheat Detection. Several communities are dedicated to cheating in Warcraft III. They produce and sell numerous cheat binaries, very rarely revealing actual source code. Since cheat-software sales are at stake, cheat authors compete to be the first to publish a cheat that works against the newest patch level of the game. Ironically, many authors steal other authors' cheat codes and techniques to spread them to new communities claiming them as their own work – there truly is little honor amongst cheaters.

Fortunately, Warcraft III was recently patched meaning a spectrum of first-to-release cheats were available for this experiment. Cheats were run in isolation by hosting non-ranked LAN games to avoid disturbing legitimate players. Fides was able to detect every cheat collected; the five most prominent cheats (all originals, not variants) being: *Bendik's MapHack* which is very minimalist and reveals no more than hidden units on the main map, *Kolkoo's MapHack* which patches code over a few pages, the *Revealer MapHack* which hot patches a small number of bytes over fewer pages but also injects code that hooks game input functions so that it may be toggled on or off, the *Simple MapHack* which hot patches a mere 61 bytes but over a number of pages, and Perma's *Nemesis MapHack* which is the prominent success of this experiment.

Perma is viewed in many communities as the foremost Warcraft III cheat author. His Nemesis MapHack (which he advertises as "undetectable") is the sequel to his infamous *Zerocraft* cheat which went undetectable for just short of two years before it was discontinued (Blizzard finally obtained a copy and developed a signature for it). In an attempt to prevent Blizzard from obtaining Nemesis, the process to obtain a copy requires one to have a profile with good standing in his home cheat community and purchase the cheat for \$25 – amazingly more than the purchase price of the game. This cheat has more features than the other cheats and is heavily obfuscated to prevent Blizzard (as well as rival cheat authors) from learning his tricks.

Figure 6 shows that the Warcraft III cheats are detected using upwards of two hundred Hash Page audits which is not surprising given that the game has 22 threads and sleeps roughly 77% of the time. At this scale one can see a trend that the number of audits required to detect a cheat tapers off. This occurs because at larger audit frequencies the inter-arrival randomness grows to the point where the randomness (between $\pm$ 5% of the frequency) surpasses the loop duration of 15.6ms so the probability of detection asymptotically approaches the probability to detect the cheat with a single random audit.

Logically, the probability to detect a cheat grows with respect to the changes that the cheat makes to the game (i.e., a cheat that modifies more code pages becomes easier to detect). To our satisfaction, this makes the feature-rich Nemesis MapHack among the easiest cheats to detect without ever reverse engineering it. The number of audits required to detect Nemesis taper off somewhere around 300, meaning it can be detected in about 5 minutes when auditing at a leisurely frequency of once per second, or in 20 seconds when auditing at a more aggressive frequency of once every 100ms. In contrast, Bendik's MapHack which makes almost no changes to the game would take 22 minutes to detect when auditing once per second, or 2 minutes to detect when auditing once every 100ms.

5. DISCUSSION

5.1 Limitations

Attacks on the Auditor. Cheaters own the systems they cheat with so they may tamper with any component to evade detection. Likely an adversary would directly target the Auditor. A number of techniques could bolster Fides against such attacks including directly accessing memory, audit entanglement, Auditor and client polymorphism, or leveraging tamper-resistant co-processors. These techniques are elaborated upon in Section 5.2.

Elementary Measurements. While the measurements presented in this paper detect current and foreseeable cheats, cheaters may evolve methods that the measurements cannot detect efficiently. It is important to note that Fides is not restricted to the four presented measurements and can be easily extended to include new measurement types.

Timing Difficulties. Network latency and jitter add timing inaccuracy at a resolution several magnitudes greater than that which code executes at, preventing any remote software integrity system (including Fides) from predicting a client's exact execution state upon receiving an audit packet. Thus such systems are forced to validate the correctness of whichever state the client is observed to be in rather than demand the client be in precisely one state.

External Cheats. Fides focuses on cheats that affect the proper execution of the game client and does not address cheating external to it. For example, online poker cheaters collude by sharing private information (i.e., the cards in their hand) to defraud legitimate players. Other cheaters employ robotic peripherals to automate repetitive or precision-based gameplay (e.g., the Guitar Hero robot [36]). Cheats that never modify the client cannot be caught by detection of anomalous client execution but instead may be caught by detection of anomalous player behavior (e.g., observing wins correlated to unusual gameplay, abnormal grouping patterns, or highly erratic player skill).

Poor Game Design. Cheat detection may not address behavior that is against the spirit-of-the-game yet is possible without modifying the client or employing external devices. For example, "player achievements" create noteworthy in-game objectives (publicized when a player accomplishes one) but when implemented improperly are trivialized by players who design custom levels to do so. Such problems should be solved by better game design, in this case disallowing players from attaining achievements while playing custom levels.

Macros and Keybindings. Many developers relax the prohibition on automation by allowing players to customize their game interfaces with macros of and keybindings for authorized game commands. Games like World of Warcraft use execution-tainting mechanisms to distinguish permitted customization from excessive automation [37]. For accurate anomaly detection, these interface customizations should be isolated to well-defined regions of client memory where they will not interfere with audits of game code or data. Indirect jumps that access them can be easily validated if the jumps are restricted into those isolated regions. Furthermore, the client must commit to the customizations (i.e., send them to the server) before using them so that the server may verify their legality and Fides may validate that they are not modified at inappropriate times during gameplay.

5.2 Future Work

There are a number of techniques that could be explored to strengthen the Fides approach (specifically the Auditor) against evasion by the cheater.

Direct Access. Locating the Auditor within the client process provides direct access to its virtual memory without relying upon system functions like `ReadProcessMemory()`. This eliminates a number of opportunities for a cheater to manipulate data collected by the Auditor.

Audit Entanglement. Fides may authenticate audited data by entangling the measurements with time-sensitive cryptographic computations, similar to Pioneer [29]. Each entangled computation would depend upon a random nonce (sent as part of the request) and be constructed such that modification to the measurement alters the correctness or timeliness of the generated result. A valid Auditor response would then be the measured data and the proper computed result, both returned within the time limit.

Auditor Polymorphism. Rather than support defined measurement types, the Auditor could instead accept and execute short auditing routines crafted arbitrarily by the Controller. The measurement types and targets may be changed surreptitiously at any time, dynamically adjusting how the client is audited. Cheaters cannot evade detection by simply using static virtualization; they must understand precisely what each audit routine is measuring and formulate coherent responses (through their own client emulation).

To effectively manipulate data collected by the Auditor, cheaters must completely interpret every audit routine and emulate legitimate client operation sufficiently to generate correct responses on the fly. This is challenging for cheaters to accomplish in a timely fashion, especially when auditor polymorphism includes audit entanglement. Cheaters who fail to completely cloak their changes or virtualize the entire system will be detected by the first measurement that cannot be spoofed, similar to how "undetectable" virtualized rootkits (e.g. BluePill [25]) are detected when unpredicted system functions (like `cpuid()`) behave erratically [24].

Auditor polymorphism would make the Fides approach more like the Warden system [20] in that frequent updates to client-side detection software keeps cheaters on guard.

Client Polymorphism. Using this technique, the game server periodically instructs the game clients to shuffle their memory layouts by rebasing loaded libraries to arbitrarily specified new locations. This changes the structure of the client without affecting legitimate operation, *dynamically adjusting how the client must be targeted by cheats.* Runtime library rebasing may be done similarly to how in-memory or reflective DLL injection [10, 31] loads libraries from within the process' memory (i.e., not from disk) at runtime.

Each time a new client layout is specified, cheaters must adjust their code accordingly, particularly code hotpatching long jumps or code overwritten during the rebasing. Cheaters attempting to manipulate measurement data must correct their emulation and system virtualization to remain consistent with legitimate clients. The server can update its emulated state before making the changes known to the clients, however, cheaters cannot respond until after they know the layout changes. Additionally this technique could reduce emulator state by dictating the same new layout for all clients using common libraries.

HARDWARE-BASED STEALTH MEASUREMENTS. The Fides system was designed so that the Auditor could leverage a hardware component within the client machine for providing tamper-resistant measurements of the game. The hardware component must be isolated from the host processor so that it cannot be affected by the system owner, yet it must have sufficient access to measure the system to determine if cheat software has compromised the game client.

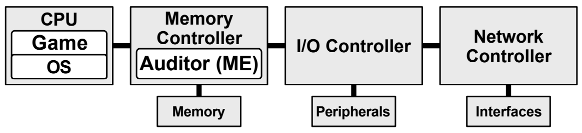


Figure 7: Functional location of one instance of Intel® AMT's Manageability Engine (ME).

The Intel® Active Management Technology (AMT) [17] platform is one such suitable hardware component. Figure 7 shows the current architecture of the AMT, specifically the location of the trusted Manageability Engine (ME). While the ME is not a full-blown CPU (i.e., it lacks the speed and features necessary to run the game client), it has been used to detect rootkits [7] and peripheral automation cheats [27]. The ME would be a good location for the Auditor because it has access to the entire contents of physical memory, it is controllable through a secure (i.e., authenticated and encrypted) network connection, and it only executes signed code assuring players that their privacy is safe.

While challenges remain to employing such a hardware component (e.g., register and memory caching, and virtual to real memory mapping), hardware support could facilitate additional measurements for detecting more subtle forms of cheating. Specifically, the addition of *memory watchpoints* would allow the Auditor to observe unusual memory accesses in terms of timing, source process, and frequency. Such a feature could reveal information exposure cheats where data is only read but not modified by external processes in an unexpected manner. Similarly, memory watchpoints could more quickly detect unauthorized modification to dynamic data by observing unusually frequent variable modification.

5.3 Related Work

ANOMALY-BASED DETECTION. Anomaly-based detection is a well explored research area. The methodology has often been applied to strong security problems like intrusion and rootkit detection [16, 18, 23, 35]. In such applications, the detector must understand the characteristics of a complex multi-faceted system. Much knowledge is required for those approaches to perfectly characterize all legitimate operation so they accept some misclassification in exchange for more manageable state. In contrast the cheating problem has a limited search space (i.e., the well-define game client) and can be efficiently learned using mostly static analysis.

The branch of research most related to the Fides approach is anomaly-based application integrity [8, 12, 28, 34] which focuses on the application's system call behavior from the point of view of a secure operating system. The "*VtPath*" method even investigates anomalies in the process stack, but only builds legitimate understanding through training and does not consider static analysis.

Some research has applied anomaly-based detection to player behavior [4, 19, 38]. This approach can detect cheats external to the game such as player collusion and "robotting" by analyzing high-level player behavior and win-loss statistics. However, this approach is very game-dependent and requires a deep understanding of the game rules, game maps, and what normal behaviors looks like. Acquiring an understanding of "normal" player behavior often requires a lot of trusted gameplay samples and can be expensive in terms of human involvement. This problem is exacerbated in persistent games (such as massively multiplayer online games) where the popularity of in-game activities change over time (sometimes in unpredictable ways) as the virtual world evolves.

TRUSTED COMPUTING. Trusted computing approaches leverage secure cryptographic software, virtual machines, or hardware components to verify that the application code is operating as intended on the remote machine. Sometimes referred to as a "*root-of-trust*," the trusted system is used as the starting point to make statements about the integrity of the application. Prominent systems include Terra [11], the TPM architecture [26], and Pioneer [29]. While these approaches quickly discover changes made to the application, they add overhead to the application's execution that may be prohibitively expensive for the real-time demands of responsive gameplay. Instead, the Auditor has fewer real-time requirements and may employ a trusted computing approach to guarantee its integrity.

PROOF-OF-CORRECTNESS. The proof-of-correctness approach focuses on cheat prevention rather than cheat detection, ensuring that the game client is running correctly [1] according to game rules and physics. This approach catches unauthorized manipulation of client state but fails to address client automation – a prevalent and disruptive form of cheating which is detected by anomaly-based detection.

ANTI-CHEATING. Many game developers use anti-cheat software: Blizzard® games (e.g., Diablo II, StarCraft, Warcraft III, and World of Warcraft) use the Warden system [20], Valve® games (e.g., Counter-Strike, Left4Dead, and Team Fortress) use the Valve Anti-Cheat (VAC) [33]. Other games (e.g., Battlefield 1942, Call of Duty, and Quake) support the use of third party anti-cheat systems like PunkBuster [6]. Unfortunately, these existing systems use signature-based detection, promiscuously scanning each and every process on the client machine leading to problems with accuracy and real attacks on the detection mechanisms [22].

6. CONCLUSIONS

Cheaters use advanced methods to manipulate games and avoid detection. This paper presented Fides, an *anomaly-based* cheat detection system that *continuously, randomly, and remotely measures* client execution. The server-side Controller specifies how and when the client-side Auditor measures the game. To accurately validate audits, the Controller *emulates the game client* and corroborates dynamic data with the server. A software prototype demonstrated that Fides is able to efficiently detect several existing cheats, including one cheat that is advertised as "undetectable". Employing techniques like audit entanglement, Auditor or client polymorphism, or placing the Auditor on secure hardware would make audits more tamper-resistant.

7. REFERENCES

[1] N. E. Baughman and B. N. Levine. Cheat-Proof Playout for Centralized and Distributed Online Games. In *IEEE INFOCOM*, April 2001.

[2] Blizzard Entertainment. Warcraft III Homepage. `http://www.blizzard.com/us/war3/`.

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

[4] K. Chen, H. K. Pao, and H. Chang. Game Bot Identification Based on Manifold Learning. In *NetGames*, October 2008.

[5] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium*, January 1998.

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

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

[8] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *IEEE Symposium on Security and Privacy*, May 2003.

[9] W. Feng, E. Kaiser, and T. Schluessler. Stealth Measurements for Cheat Detection in On-line Games. In *NetGames*, October 2008.

[10] S. Fewer. Reflective DLL Injection, 2008. `http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf`.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. *SIGOPS Operating System Review*, 37(5), 2003.

[12] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security*, 6, August 1998.

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

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

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

[16] H. Inoue and S. Forrest. Anomaly Intrusion Detection in Dynamic Execution Environments. In *New Security Paradigms Workshop*, September 2002.

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

[18] C. Kruegel and G. Vigna. Anomaly Detection of Web-Based Attacks. In *ACM CCS*, October 2003.

[19] P. Laurens, R. F. Paige, P. J. Brooke, and H. Chivers. A Novel Approach to the Detection of Cheating in Multiplayer Online Games. In *IEEE ICECCS*, 2007.

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

[21] National Institute of Science and Technology. Secure Hash Standard, April 1993.

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

[23] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - A Coprocessor-Based Kernel Runtime Integrity Monitor. In *USENIX Security Symposium*, August 2004.

[24] T. Ptacek and N. Lawson. Don't Tell Joanna, The Virtualized Rootkit is Dead. In *Black Hat*, August 2007.

[25] J. Rutkowska. Subverting Vista Kernel for Fun and Profit: Part 2 - Blue Pill. In *Black Hat*, August 2006.

[26] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-Based Integrity Measurement Architecture. In *USENIX Security Symposium*, August 2004.

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

[28] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *IEEE Symposium on Security and Privacy*, May 2001.

[29] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. In *ACM SOSP*, October 2005.

[30] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *ACM CCS*, October 2004.

[31] Skape and JT. Remote Library Injecion, 2004. `http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf`.

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

[33] Valve Software. Valve Anti-Cheat. `https://support.steampowered.com/kb_article.php?p_faqid=370`.

[34] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, May 2001.

[35] K. Wang and S. J. Stolfo. Anomalous Payload-Based Network Intrusion Detection. *LNCS*, 3224, October 2004.

[36] Wired News. Guitar Hero Robot Plays Videogame With Electronic Precision. `http://blog.wired.com/gadgets/2008/11/guitar-hero-rob.html`.

[37] WoWWiki. Secure Execution and Tainting. `http://www.wowwiki.com/Secure_Execution_and_Tainting`.

[38] S. Yeung, J. Lui, and J. Yan. Detecting Cheaters for Multiplayer Games: Theory, Design and Implementation. In *CCNC*, January 2006.