Extracting Threat Intelligence From Cheat Binaries For Anti-Cheating

Md Sakib Anwar
anwar.40@osu.edu
The Ohio State University
USA

Chaoshun Zuo
zuo.118@osu.edu
The Ohio State University
USA

Carter Yagemann
yagemann.1@osu.edu
The Ohio State University
USA

Zhiqiang Lin
zlin@cse.ohio-state.edu
The Ohio State University
USA

ABSTRACT

Rampant cheating remains a serious concern for game developers who fear losing loyal customers and revenue. While numerous anti-cheating techniques have been proposed, cheating persists in a vibrant (and profitable) illicit market. Inspired by novel insights into the economics behind cheat development and recent techniques for defending against advanced persistent threats (APTs), we propose a fully automated methodology for extracting "cheat intelligence" from widely distributed cheat binaries to produce a "memory access graph" that guides selective data randomization to yield immune game clients. We have implemented a prototype system for Android and Windows games, CheatFighter, and evaluated it on 86 cheats collected from a variety of real-world sources, including Telegram channels and online forums. CheatFighter successfully counteracts 80 of the real-world cheats in under a minute, demonstrating practical end-to-end protection against widespread cheating.

KEYWORDS

Anti-cheating; Program analysis; Automated client hardening

ACM Reference Format:

Md Sakib Anwar, Chaoshun Zuo, Carter Yagemann, and Zhiqiang Lin. 2023. Extracting Threat Intelligence From Cheat Binaries For Anti-Cheating. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3607199.3607211

1 INTRODUCTION

One of the largest entertainment industries today is the computer game industry, which is projected to reach a market size of $200 billion by 2024 [3]. Currently, mobile games account for 52% of the gaming market, followed by console games at 28%, and PC games at 20% [3]. Unfortunately, where there are games, there are also incentives to build and sell cheats for profit. Cheaters often seek to modify game-critical variables to achieve unfair advantages such as "god mode" or infinite ammo.

Prior studies show that 37% of players have admitted to cheating at some point [26], and benign players who have had a negative experience due to cheating are 48% less likely to buy in-game items and 77% more likely to stop playing the game altogether, leading to lost revenue [1, 14]. Furthermore, various underground markets (e.g., UnKnoWnCheaTs [8]) fuel rampant cheating by selling prepackaged cheats and relevant knowledge.

In response, numerous defenses have been proposed, including operating system security modules (e.g., Security Enhanced Linux [7]), program hardening and obfuscation [6], and server-side anomaly detection [13]. Unfortunately, despite these prior proposals, cheaters continue to find workarounds to exploit new weaknesses to the detriment of game studios and publishers.

Interestingly, we observe that while game cheating appears at first glance to be analogous to other forms of hacking, there is a unique economic factor at play in this illicit ecosystem that does not exist in other contexts. Specifically, in order for cheat developers to attract customers, the prepackaged cheats they sell must work repeatedly, not just once like in the case of attacks launched by advanced persistent threats (APTs). Conversely, games are easy for defenders to update thanks to centralized digital storefronts like Steam and Google Play. This creates a unique opportunity to automatically and rapidly counteract cheats sold on underground markets. *Unfortunately, this opportunity is currently unexplored because existing defenses are untargeted, making no use of sold cheat packages.*

In light of this, we propose an agile and automated defense that aims to cripple the underground market by automatically diversifying game clients based on *intelligence extracted directly from the sold prepackaged cheats.* Specifically, we propose to recover the payloads from prepackaged cheats to guide targeted data structure randomization in game clients to efficiently thwart widespread cheating. Our solution is fully automated and does not require any human in the loop, so defenders can cut off attack vectors faster than cheaters can monetize them. In essence, we are turning the underground market's own efficiency against itself.

RAID '23, October 16–18, 2023, Hong Kong, China
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0765-0/23/10…$15.00
https://doi.org/10.1145/3607199.3607211

Notice how our proposed approach does not aim to make cheating impossible. Instead, our objective is to eliminate the economic incentives that can turn cheating into a widespread issue that drives away benign players in droves. Furthermore, the solution we propose is unique in that it is a *payload-oriented* analysis of attack payloads, as opposed to typical exploit analysis that focuses on the *vulnerability*. This difference reflects our goal of removing economic incentives as opposed to patching a buggy piece of code.

In designing our solution, we quickly realized that traditional methodologies such as dynamic analysis and sandboxing (seen typically in malware analysis) are not suitable for achieving our vision. First, like malware, cheats can employ evasive tactics, including delayed execution and sandbox detection. Second, unlike malware, cheats are often context-sensitive. For instance, a cheat may change an ammo value only when a particular weapon is wielded by the cheater. Furthermore, to make our solution practical and safe for developers, our design should minimize assumptions about the execution environment and avoid invoking adverse effects that could arise from executing the cheat blindly. Therefore, we opt for a static approach to intelligence extraction that then carefully guides a subsequent dynamic analysis.

Due to the unique constraints surrounding our proposed defense, we had to identify and overcome several technical challenges specific to our payload-oriented analysis. *First,* cheats are typically distributed as stripped native binaries, even when the target is a mobile game. In such cases, our system must be able to analyze binaries that mix native instructions and interpreted bytecode. *Second,* our solution must identify only the logic related to performing the cheat and ignore extraneous capabilities like evasion. *Third,* in order to effectively leverage the statically extracted intelligence about the cheat to guide targeted data randomization, our system must be capable of mapping its findings back to source code, bridging several semantic gaps throughout the game's code.[1]

With these criteria in mind, we present CheatFighter, a defense prototype for Windows and Android games that uses 1) an API-aware data dependency-based algorithm to *extract intelligence* from cheat packages, 2) dynamic analysis to *translate extracted intelligence* to source code, and then 3) source code analysis to selectively *randomize* the sensitive variables targeted by cheaters. CheatFighter operates as a fully automated standalone tool that developers can directly incorporate into their existing pipeline to (re)harden game clients.

We have evaluated CheatFighter using 86 real-world cheats collected from real sources that include Telegram channels, online forums, and cheat sharing websites, targeting popular online games like Call of Duty and PUBG for Windows and Android. CheatFighter successfully extracts the payloads of 80 cheats and counteracts them with selective data randomization, without any developer assistance. We manually validated that the automatically applied countermeasures yield no observable impact on the game's performance or functionality. Our evaluation shows that it takes CheatFighter less than a minute on average to analyze a cheat and apply changes to the client for defense. We have made our code publicly available at: https://github.com/OSUSecLab/CheatFighter.

[1]Most production game engines leverage automatic language transformation (e.g., C# to C++) and ahead-of-time (AOT) compilation.

Table 1: Post Classification for Cheat Sharing Website

Attack Surface	Cheat Category	Post Type			
		Release		Coding	
		#	%	#	%
Client	**Memory Modifying**	183	40.0	94	65.0
	Static Code Patching	53	11.6	5	3.5
	ShellCode	71	15.5	15	10.3
Unidentified	**Unidentified**	152	33.1	31	21.4
	Total	**459**	-	**145**	-

2 OVERVIEW

2.1 Cheat Taxonomy

Cheating techniques for computer games can be divided based on the following attack surfaces: (1) server-side cheats, (2) network channel cheats, and (3) client-side cheats [35, 51]. Examining the first two categories, server-side cheats exploit vulnerabilities in the game servers [29, 45] whereas network channel cheats exploit communication weaknesses between the server and client, such as missing integrity checks [54]. Notice how these categories rely on exploitation techniques that are not specific to computer games. Consequently, they can be mitigated with existing defenses [30, 46].

Conversely, defending against client-side cheats is significantly harder, as such attacks occur on player computers outside the game developer's direct control. Client-side cheats can be further subdivided into three types: *bots*, *static code patching*, and *dynamic memory modification.* Bots aim to automate redundant player tasks, such as mining in-game resources, and can be mitigated with anomaly detection techniques [18, 47]. In contrast, static code patching and dynamic memory modifications directly target the game's code and runtime state, making them harder to mitigate.

Targeted Cheats. Prior work [49, 53] has identified memory modifying cheats as one of the biggest threats to gaming. Furthermore, we have scraped a popular cheat sharing website and classified posts using term frequency–inverse document frequency (tf-idf). Based on our results presented in Table 1, 40% of the posts under the Release prefix and 65% of the posts under the Coding prefix are related to memory modifying cheats, making it by far the most prevalent type.[2] *In summary, due to their prevalence and difficulty to mitigate, we focus on memory modifying cheats for our work.*

2.2 Cheat Development

Attacker Model. Our threat model considers an adversary or cheat developer who can fully compromise the client device, meaning that they can escalate their privileges and alter the way the OS communicates with the game. However, unlike other attacks, this scenario requires the adversary to design an attack that can be replicated across multiple devices and instances. This imposes a unique constraint on the adversary regarding the vulnerabilities they can exploit to create a successful cheat. Nevertheless, we have

[2]Some posts could not be classified with high confidence based solely on keyword tf-idf. We label these as Unidentified to be conservative.

```
1  int main(int argc, char** argv) {
2      ..
3      char *package = "com.tencent.tmgp.sgame";
4      pid = getPID(package);
5      sprintf(mem_file_name, "/proc/%d/mem", pid);
6      sprintf(map_file_name, "/proc/%d/maps", pid);
7      memfd = fopen(mem_file_name, "r");
8      library_name = "libil2cpp.so";
9      base_address = get_module_base((char *) &library_name);
10     add = base_address + 0x820cc24;
11     lVar1 = readValueL(add);
12     add2 = base_address + 0x50;
13     lVar1 = readValueL(add2);
14     add3 = lVar1 + 0x8;
15     ..
16 }
17 ulong get_module_base((char *) &library_name) {
18     ..
19     FILE *mapping = fopen(map_file_name, "r");
20     ..
21     char *pcVar1, *base_address_string;
22     ulong base_address; char file_content[1024];
23     do{
24         pcVar1 = fgets(file_content,0x400,local_18);
25         ..
           pcVar1 = strstr(file_content, library_name);
26     } while (pcVar1 == (char *)0x0);
27     base_address_string = strtok(file_content,"-");
28     base_address = strtoul(base_address_string,
29         (char **)0x0,0x10);
30     return base_address;
31 }
32 undefined8 readValueL(__off64_t param_1) {
33     ..
34     pread64(memfd,&local_8,0x4,param_1);
35     return local_8;
36 }
```

Figure 1: Code Excerpt from a Decompiled Cheat (Android)

observed adversaries assuming similar capabilities (e.g. full control, privilege escalation) on the user's side as well.

Cheat developers typically overcome modern defense mechanisms (e.g. ASLR) in locating address of critical variable by calculating addresses with respect to some stable references and by traversing object pointers. By applying both static and dynamic analysis to the game using various tools (e.g., [2, 9, 16, 25]), the cheat developer can discover a path to the critical variable and use it to access the critical data. In particular, a cheat developer has to overcome three obstacles:

Access to Game Memory. A memory modifying cheat must access the memory of another process overcoming the OS' memory isolation mechanisms. Cheats typically run as root or employ virtual environments [4] in order to access debugging APIs (e.g., `ptrace` in Linux, `KERNEL32` APIs in Windows) or directly read the game's memory (i.e., `/proc/$pid/mem` in Linux).

Path to Critical Variable. A cheat developer finds the critical variable in memory by searching for known values and making changes to them using dynamic analysis tools. Once identified, a more robust way of reaching the critical variable is needed that does not rely on trial and error. The most reliable way of doing this is by finding a chain of pointers leading to the critical variable, starting from fixed reference point (e.g. base address of libraries mapped into memory). However, without access to the source code, a cheat developer has to extensively test discovered pointer chains to ensure that they work every time. Thus, finding a working pointer chain is extremely tedious and laborious.

Creating Cheat. Finally, cheat developers need to develop a cheat with this pointer chain that can inspect the runtime memory of the game to reach the target variable. Depending on the target platform, architecture, and even the compilation method used, this can be achieved in multiple ways. Nevertheless, as cheats need to use debugging APIs, they are often implemented in low-level languages (such as C) and compiled to run on a specific target device.

2.3 Motivating Example

To illustrate exactly how a cheat works, Figure 1 shows a piece of code decompiled from a real-world Android cheat. The cheat code is part of an APK (often an assest) that can run as a separate program. Once the cheat binary is copied to a directory with execution privilege and executed by the APK, the first thing the cheat does is find the `pid` of the game. This can be done by looking through the `/proc` directory and reading the `cmdline` file under each process to search for the package name of the target game. For brevity, we have excluded this code from the figure.

Once found, the cheat has all the information it needs to read and modify the game memory. In particular, Figure 1 shows two helper functions: `get_module_base`, which retrieves the base address of a given referencing name[3] by parsing the content of the memory map file, and `readValueL`, which reads from a specific location in the process memory. We can see on line 34 that a `pread64` call has been made on a file descriptor `memfd`, which was initialized with the "`/proc/$pid/mem`" file (line 7). Finally, the `param_1` in the function acts as an offset for the call to `pread64`, which allows the cheat to read from a certain offset of the memory. A full graph representation of all the dereferences and offset arithmetic performed by this cheat program is shown in Figure 2.

2.4 Challenges & Insights

From the example shown in Figure 1 and discussed in §2.3, we can see that recovering the chain of pointers used by a cheat binary in order to guide a defense like data structure randomization is challenging due to the lack of context or references within a stripped cheat binary. Gathering the scattered intelligence and representing them in a unified form to be used as a guide for game client hardening raises several technical challenges:

Locating Pointer Heads. As seen in Figure 1, the head of the pointer has two defining identities: a referencing name and the runtime base address of the referencing name in the memory of the game. For example, in Figure 1, the variable `base_address` and the referencing name `libil2cpp.so` collectively form the head of the pointer. Since the value of `base_address` will change with each execution due to dynamic resolving by the OS loader, our analysis must retain the context of `libil2cpp.so` to robustly identify the pointer chain.

[3]We use the term "referencing name" to mean named aliases in a broad sense. These include symbols in the program binary and filenames in process memory maps.

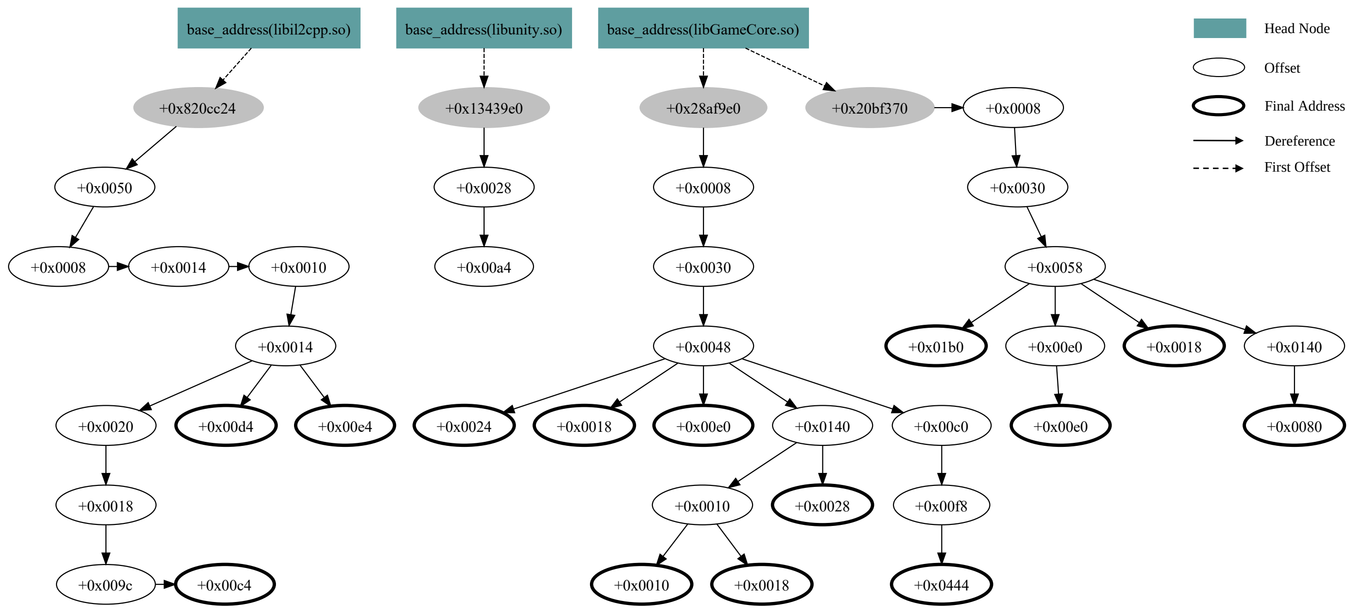

Figure 2: Memory Graph Used in a Cheat for Arena 5v5.

With a systematic investigation of current cheat development practices, we noticed that these base address variables can be identified using an API-aware data flow analysis. For example, in Linux, programs must rely on the information in the "`/proc/$pid/maps`" file. The operations done to this file to extract the required information can be modeled into a series of API calls. In Windows, cheat programs use the base address of the process itself as the reference. Specifically, cheats use KERNEL32 APIs to find the game process by name and subsequently its base address. In both cases, an API-aware data flow analysis, starting from either the "`/proc/$pid/maps`" file or the name of the game process, can locate the variable holding the base address.

Identifying Valid Offset & Memory Access. It is *not* the case that any arbitrary addition operation will eventually lead to a memory access in the pointer chain. Furthermore, it is not guaranteed that the offset is a single value or computed in one step. For example, to get a correct offset, several values may be added to the current pointer. Therefore, the challenge when observing a value being added to the current address is deciding whether it is the final offset.

Our insight for dealing with this challenge is to focus on the value's usage because no matter how the valid offset is generated, it must eventually be used to access the game's memory (e.g., dereferencing the data). Such memory accesses are often achieved via particular function calls (e.g., `read`). For instance, the offset `0x820cc24` in line 10 has been used to perform a dereference at line 11 by using the function call `readValueL`, which subsequently uses `pread64` at line 34. Furthermore, as evident from line 10 and line 12, the same address might be used in multiple further address calculations later in the cheat program's execution.

Mapping Back to Source. Although the cheat program contains intelligence about the target game's internal workings in the form of the pointer chains it uses, these chains only have meaning once considered in the context of the game's memory state. In other words, to fully utilize the cheat program in protecting the game, these addresses and offsets need to be resolved into symbols (data structure names) and offsets so that the hardening step in our defense can correctly identify the targeted variables. However, this is easier said than done since the code a developer writes goes through various code transformations within the game engine. For example, in the popular Unity game engine, developers write code in C#, which is then automatically transformed by the engine into C++, bundled with additional assembly code, and then compiled and stripped into a shared library. This ahead-of-time compilation is not only efficient, but also necessary for resource constrained platforms like Android. The final executable is then mapped to the process memory, where cheat binaries then target critical variables using their reverse engineered pointer chains. Thus, a challenge emerges in bridging the semantic gap between the pointer chains used by the cheat program to target runtime memory variables in the game's process and the original source code representation of those variables. Without the latter, any attempts to randomize data structures would be best effort and potentially game breaking. In short, this challenge boils down to a type inference memory forensics problem.

Also notice that since the cheat's pointer chains target a particular version of the game, our solution cannot rely on modifying the game client to make the analysis easier, as this would break the pointer chain and with it, the true semantics. Thus, our solution must analyze the cheat in the context of a production version of the game client, which is itself stripped. Interestingly, we notice that we can use reflection properties of base data structure (e.g. Object in Java, System.Object in C#) to find the class name and subsequently the target field, with the help of the recovered final offset.

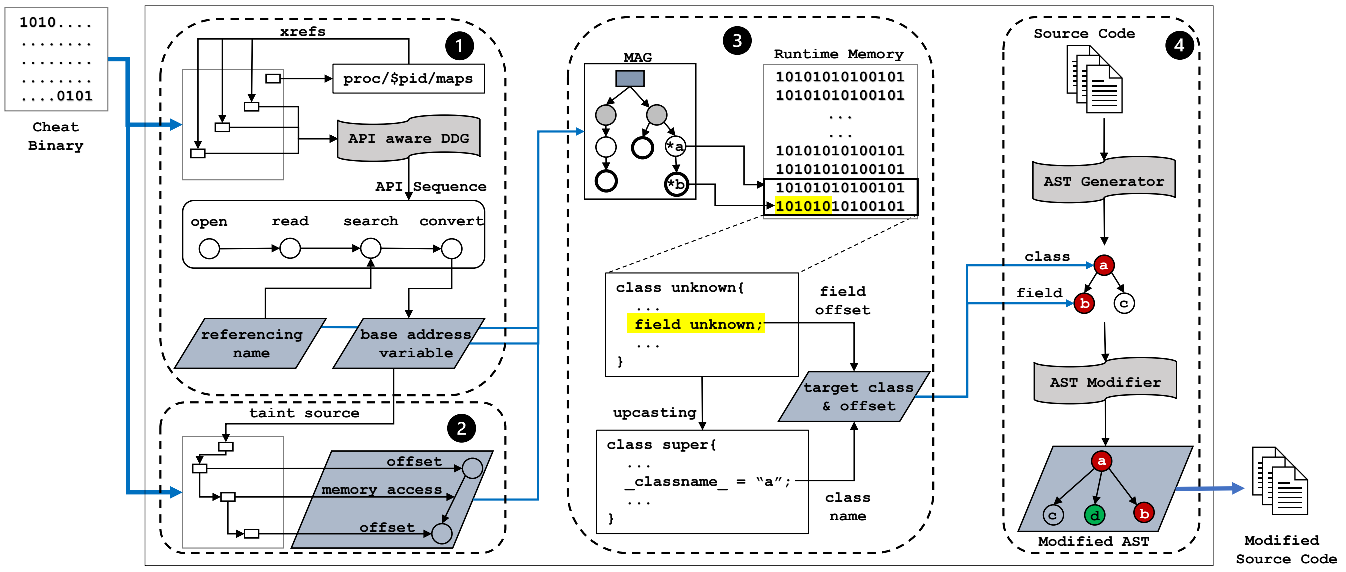

Figure 3: CheatFighter Design. Note that ❶ denotes Head Node Identification, ❷ Children Node Discovery, ❸ MAG Translation, and ❹ Selective Client Obfuscation

Automating Client Obfuscation. Field offset randomization for security hardening is a well explored topic that has even seen recent adoption within the Linux kernel via RANDSTRUCT for Clang. However, current state-of-the-art frameworks require the developer to specify where to apply randomization via a special keyword, and require use of a specialized compiler that supports this keyword. Since compiler modification is required, current work focuses on only a subset of languages, namely C++.

Unfortunately, game developers typically use higher level languages like C#, not C++, and game engines employ proprietary compilers that are not easy to modify. These factors combined make current off-the-shelf randomization solutions poorly suited for our usage scenario.

However, if our solution can identify targeted variables at source code level (see previous challenge), then it can apply randomization directly to the source's abstract syntax tree (AST), prior to compilation. This would remove any reliance on the developer being in the loop, and avoid any compatibility issues with their existing compiler toolchain. Additionally, our solution can add padding fields to ensure that simple signature searches for critical data structures in memory will fail, further thwarting cheat programs.

Unified Representation of Intelligence. Finally, although we have described how to identify the cheat intelligence in a cheat program, we still require a unified representation for all these scattered bits of reverse engineered knowledge implemented by the cheat developer. To this end, we propose an encoding that we call the Memory Access Graph (MAG), which aims to retain the context (referencing name), pointer chains (both offset additions and memory accesses), and branching in the chains in an easy to understand and visualize format. Figure 2 shows the MAG for a real-world cheat program, where each root node is the head of a pointer chain and the leaf nodes are the game variables targeted by the cheat.

3 DESIGN

We have addressed the aforementioned challenges with the insights we have with respect to MAG, and designed CheatFighter, a binary analysis tool that can extract MAG from a cheat binary and translate it with an API-aware, context-sensitive, data dependence analysis. At a high level, as explained in §2.4, CheatFighter is composed of four components, each of which addresses one challenge:

❶ **Head Node Identification** (§3.1). It scans the cheat binary code to locate each API by following the control flow graph. Next, by matching the series of APIs with predefined patterns, it locates the code that is being used to extract the head of the pointer. Furthermore, it connects the variable holding the base address to the referencing name as to retain context, both of which together forms the head node of the MAG.

❷ **Children Node Discovery** (§3.2). It launches context-sensitive data dependency analysis that takes each head node as a source. It tracks the accumulated offsets and memory access to create children nodes in the MAG where each edge represents a memory access and each node an offset addition and ultimately an address. While building the MAG it simultaneously keeps track of all the sub-paths, which are combined to create the complete MAG.

❸ **MAG Translation** (§3.3) It runs a dynamic analysis of the game and reveals the data structures associated with each node of MAG guided by the reflection properties of the language the games are written in.

```
address            perms ...  pathname
85880000-8d359000  r--p  ...  /data/app/.../libil2cpp.so
8d359000-8df92000  rw-p  ...  /data/app/.../libil2cpp.so
8ec1a000-8ec1b000  rw-p  ...  /data/app/.../libil2cpp.so
8ec1b000-8ec20000  r--p  ...  /data/app/.../libil2cpp.so
8ec20000-8eca2000  rw-p  ...  /data/app/.../libil2cpp.so
6b53b000-6db54000  r--p  ...  /data/app/.../libGameCore.so
6db54000-6e0d5000  rw-p  ...  /data/app/.../libGameCore.so
6e583000-6e584000  rw-p  ...  /data/app/.../libGameCore.so
6e584000-6e589000  r--p  ...  /data/app/.../libGameCore.so
6e589000-6e60b000  rw-p  ...  /data/app/.../libGameCore.so
06000000-07173000  r--p  ...  /data/app/.../libunity.so
07173000-071b6000  rw-p  ...  /data/app/.../libunity.so
```

Figure 4: Memory Mapping of Arena 5v5 in Android

❹ **Selective Client Obfuscation** (§3.4) It takes in the names of the target variable and the source code to generate and modify the ASTs to selectively obfuscate the code by randomizing existing fields and by adding padding fields.

In the interest of clarity, we will use Android as the running concrete example to explain CheatFighter's design and our solutions. However, our approach generalizes to other systems, which we demonstrate by also implementing support for Windows and evaluating Windows-based cheats. We elaborate on the effort required to add support for additional OS in §3.5.

3.1 Head Node Identification

Identifying the head node of the MAG means identifying the variable that holds the base address and finding the referencing name together, which can be read from the memory map file. The memory map file contains information about the shared objects that have been mapped in the memory of the process. Figure 4 presents part of the memory map file for the game `Arena 5v5`. Each line of the file shows information related to a shared object. Among them, there are two important pieces of data: 1) the *name* of the shared object (i.e., the referencing name), such as `libil2cpp.so` for the first line; 2) the *addresses* of the memory region to which the shared object has been mapped, such as `0x85880000` to `0x8d359000` for `libil2cpp.so`. Therefore, the base address (e.g., `0x85880000` for `libil2cpp.so`) of a shared object can be found by parsing the content of the memory mapping file. In order to extract the cheat intelligence, CheatFighter needs to identify the variable holding the base address of the shared object and its referencing name, detailed below.

Base Address Identification. As mentioned earlier, this can be modeled as the output of a series of operations made on the mapping file and its content. More specifically, CheatFighter needs to identify the following operation as modeled by APIs: 1) opening the mapping file (e.g., `fopen`); 2) reading the content from the file (e.g., `fgets`); 3) searching for a particular line (e.g., `strstr`); 4) extracting a substring and converting it to a number (e.g., `strtoul`). Since the cheat is often developed with low-level programming languages (e.g., C/C++), APIs are used to open, parse the memory map file, search the shared object of interests, and obtain their base addresses. Through systematic enumeration of all possible APIs, we have identified 20 of them that are relevant, as shown in Table 2. The parameters in bold are used to link them together based on the

Table 2: The APIs involved in the identification of base addresses

Operation	Function	Synopsis
File Open	fopen	fopen(***pathname**, mode);
	open	open(***pathname**, flags, ...);
File Read	fgets	fgets(buf, size, **stream**);
	gets	gets(***stream**);
	fscanf	fscanf(**stream**, format, ...);
	read	read(**file_descriptor**, ...);
	fgetc	fgetc(***stream**);
	fread	fread(ptr, ... ,**stream**);
	fwscanf	fwscanf(***stream**, format);
	vfscanf	vfscanf(***stream**, fmt, ...);
String Search	strstr	strstr(***haystack**, *needle);
	strcasestr	strcasestr(***haystack**, ...);
	memcmp	memcmp(***str_1**, ***str_2**, n);
	regexec	regexec(*preg, **str**, ...);
	strcmp	strcmp(***str_1**, ***str_2**);
	strncmp	strncmp(***str_1**,***str_2**, n);
String to Number	strtoul	strtoul(**number_pointer**, ...);
	atoi	atoi(***number_pointer**);
	sscanf	sscanf(... , format, ...);
	atol	atol(***number_pointer**);

data dependence. The base address variable is the return value of the last API call.

For example, in our running example (line 19 in Figure 1), we notice that the `mmap` file path string (i.e., `/proc/$pid/maps`) is the parameter for the file opening API call, which is required. Meanwhile, the string itself is not a common string, as it has aliases such as `/proc/%s/maps` and `/proc/self/maps`. Therefore, CheatFighter starts the analysis by looking for the path string of the memory map file and all its aliases. Once such a variable (so-called *varnode* in `Ghidra` PCode) is identified, CheatFighter then keeps track of the varnode in the data dependency graph (DDG) and records the APIs encountered. During the process, if an API sequence that matches Table 2 has been found, the base address variable is identified, which is typically the output of the last API call (e.g., `strtoul` on line 28). This whole process can be seen in ❶ of Figure 3 where the analysis starts from the mapping string and gives the result in the form of *base address variable* to be used in the next step, children node discovery (§3.2).

Referencing Name Discovery. Although the base address variable has been identified from the cheat binary, referencing name is still unknown and is required to denote a head node in MAG. To this end, we need to find the concrete referencing name of the corresponding base address variable. For instance, in the running example, CheatFighter needs to find the referencing name (i.e., `libil2cpp.so`) for variable `base_address`.

Meanwhile, even though the base address variable is connected with referencing name in the semantic level (i.e., the base address is the runtime base address of referencing name in memory) syntactically they are not part of same instruction, API call or even function body. But from the running example, we can clearly see that they can be connected by the API call series. In particular, the base address variable is the output of the series, and referencing name is the input of the API call in the string search step because referencing name is used to find the target line in the memory map file. As such, by tracing the inputs of that API call, the referencing name can be located. As presented in Table 2, there are several API calls that can be used to search for strings. We have marked the parameter that it is looking for, which can potentially contain the referencing name. This can be seen in ❶ of Figure 3 where the referencing name is the input of the search API in the sequence.

3.2 Children Node Discovery

Once the head node is discovered, CheatFighter needs to identify the nodes (offsets) and the edges (memory accesses) of the MAG. As mentioned in §2.4, CheatFighter uses a sub-path extraction algorithm to identify the related instructions to extract the children nodes. To be more specific, while the nodes are being discovered, it is often not enough to keep an eye on the latest node or the leaf node, since the current state (sub-path) of a MAG can be stored in a local or global variable for later use. This is why for every node discovered so far, we need to maintain two things: 1) a set of registers or variables (either local or global), i.e., *pointers* holding reference to the node, and 2) a *sub-path* representing the path from the root node to the current node. As we analyze more instructions, we identify more operations, and we keep adding or updating the sub-paths.

However, to maintain accurate data dependence, we need to resolve the indirect references (local or global variables) i.e. we need to carry out pointer analysis. Existing works on static pointer analysis are often performed on the whole program at the beginning of the analysis, which often adds unnecessary overhead. We have employed an on-demand in-place pointer analysis using backward slicing for that purpose. Algorithm 1 presents an overview for the discovery of children nodes. As can be observed, the first step is to taint the base address variable (line 4) and to add a sub-path for the referencing name against the base address variable in SPS (line 5). Once this initialization is done, the algorithm can start tracking operations to identify offsets and memory accesses.

Offset Tracking. Tracking offset means tracking integer addition on any node(s) (on set of associated variables) of the sub-path constructed so far. The offset might not always be a literal constant, and recursive pointer resolutions are needed to find the proper offset that is being added. Yet, once an offset addition is identified, we notice that the offset can fall into the following cases:

(1) **Literal value**, in which there is no need to further analyze the offset. This offset can be used to update the data dependence of the output variable to reflect the offset addition.
(2) **Result of a series of arithmetic operations** on different integers. These operations are built upon each other, and when all the values in the expression are literals, the final value can be calculated.
(3) **Local variable** holding the concrete value in which a pointer analysis is triggered to identify the latest value held by that particular variable.
(4) **Global variable** that has been initialized before. In this case, we need to perform pointer analysis.

Algorithm 1 Sub-path Extraction (SPE)

```
 1: Global Variables: $MMA$: memory accessing APIs; $SYSCALL$: input/output mapping for
    system calls; $MAG$: Memory Access Graph
 2: Local Variables: $SPS$: sub-path pointer set; $Tainted$: set of tainted variables; $AO$: accumulated
    offset
 3: procedure SPEA($base, baseName, address$)
 4:     Tainted.add($base$)
 5:     SPS.addHeadNode($base, baseName$)
 6:     TrackOperations($address, Tainted, SPS$)
 7:     MAG.addSubPath($SPS$)
 8: procedure TrackOperations($address, Tainted, SPS$)
 9:     for ($pCode$) $\leftarrow$ GetPCodes($address$) do
10:         $inputs \leftarrow$ GetInput($pCode$)
11:         $output \leftarrow$ GetOutput($pCode$)
12:         switch pCode do
13:             case DataArithmetic
14:                 if $inputs \in$ Tainted then
15:                     if $pCode$ is AdditionOperation then
16:                         $offset \leftarrow$ FindConcreteOffset($pCode$)
17:                         AO.add($offset$)
18:                     Tainted.add($output$)
19:                 else
20:                     Tainted.removeIfPresent($output$)
21:             case BranchingOperation
22:                 $branchingAddress \leftarrow$ GetBranchingAddress($pCode$)
23:                 TrackOperations($branchingAddress$)
24:                 if IsUnconditional($pCode$) then return
25:             case DataMovementOperation
26:                 $pointer \leftarrow$ ResolvePointer($inputs$)
27:                 Tainted.removeIfPresent($pointer$)
28:                 if $input \in$ Tainted then Tainted.add($pointer$)
29:             case CallOperation
30:                 $api \leftarrow$ GetAPI($pCode$)
31:                 if $input \in$ Tainted then
32:                     if $api \in$ MMA then
33:                         $memAccessType \leftarrow$ GetMemoryAccessType ($api$)
34:                         $subpathPointer \leftarrow$ SPS.find($inputs$)
35:                         $subpathPointer$.addOffset($AO$)
36:                         $subpathPointer$.memoryAccess($memAccessType$)
37:                     else if $api \in$ SYSCALL then
38:                         $output \leftarrow$ SYSCALL.getOutput($api$)
39:                         Tainted.add($output$)
40:                     else
41:                         $entryPoint \leftarrow$ GetEntryPoint($api$)
42:                         TrackOperations($entryPoint$)
43:     if $address$.next() $\neq$ null then TrackOperations($address$.next())
```

(5) **Read from file** during runtime, which is the only case where we cannot resolve the offset, as the content of the file depends on both the runtime environment and the game itself.

Memory Access Identification. To access the memory, cheats have to use API calls and there are only a handful of them which can be traced to identify memory accesses. However, the primary question is how to access the memory of a game process from the cheat. This can be done via a number of ways such as:

(1) **Debugging APIs**: Most modern day OS ships with debugging APIs which can be used to observe the memory of a running process for debugging purpose, for example `ptrace` in Linux, which can be used to access memory of another process.
(2) **Memory File**: Fortunately for cheat developers in Linux everything is a file even the memory of a process which can be found under "`/proc/$pid/mem`" file which can be opened and read by any process that possesses the root privilege, which can be done in Android.
(3) **Special APIs**: In Linux, there are special APIs such as `process_vm_readv` which lets anyone with special privilege directly read from an offset of another process's memory.

```
...
iVar1 = _process_vm_readv_syscall;
if (param_7 != false)
{
   iVar1 = _process_vm_writev_syscall;
}
syscall((long)iVar1,(ulong)(uint)param_1,param_2,param_3,
         param_4,param_5,param_6);
...
```

Figure 5: Excerpt from a Decompiled Cheat

Thus, by tracing tainted input to any of these APIs we can identify memory access made by a cheat. However, the goal of the cheat is to read the memory at a certain offset which can be done by opening a file (`fopen`) and then seeking (`lseek`) or using APIs that take offset as a parameter such as `pread64` or `process_vm_readv`. From our analysis we have seen nearly all the cheats take the second option. Thus, specifically tracking calls to these functions with tainted variable will give us memory access operation performed on the nodes of the MAG. This step can be seen in the algorithm under the call opcode handling, where the first node on which the operation is executed is identified and then finally updated (line 34 - 36)

However, in some cases, we have seen the use of the function "`syscall`" to call the aforementioned functions to access the memory. It works by associating an integer number with each `syscall` such as `pread64` is represented by the number 0x43, so instead of calling `pread64` a cheat might call `syscall` with this value. For instance Figure 5 shows a cheat calling `syscall` with global variables that contains the necessary number. This approach works to avoid detection by anti-cheat process monitors that detect the use of certain functions on the target process. CheatFighter has handled these calls for ARM64 currently, since these numbers are architecture dependent.

An overview of this component's process of finding children nodes by analyzing each type of PCode while simultaneously maintaining an active set of sub-paths has been shown in ❷ of Figure 3. Which starts its work with the base address variable and via recursive taint analysis through all possible instruction constructs the sub-paths piece by piece and then joins them to produce the whole MAG which is the output of this component.

3.3 MAG Translation

As mentioned in §2.4, the code a developer writes goes through various forms of transformation which often even includes lifting from one language to another, since most game engines follow *"write once compile anywhere"* rule. C# being one of the most popular languages for game development, most game engines let developers write code in C#. However, it is not feasible to be run on other platforms such as Android and thus the games engines need to either 1) run the code in a virtual machine inside the target device (e.g. Mono) or 2) lift the code to a compatible one and pre-compile it (Il2Cpp by Unity). For game industry we have seen both in practice; however, for Android we have seen the second one (AOT compilation) being used more recently. Thus, mapping the addresses to the source code becomes a challenging task especially for low resource platforms like Android.

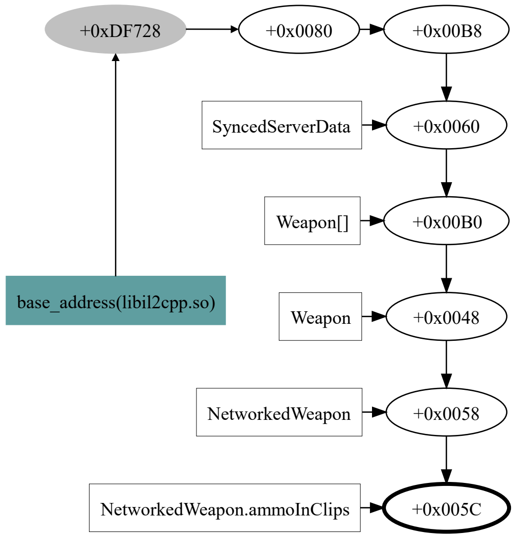

Figure 6: Translated MAG for an Open Source Game

We first attempted to map the addresses back to the compiled executable, as the compiled executable is loaded directly into the memory of the process and is the closest to the MAG. However, game engines compile not only the code a developer writes, but also many engine-related components such as game physics. Furthermore, it strips and obfuscates the code in every step and adds automated checks so that developers do not have to worry about common programming errors. All these make analyzing the executable extremely difficult, and furthermore, even if one can map the first pointer of the MAG to the memory, there is no way to know what data structure it points to. To be more specific, the binary is static, whereas MAG and the addresses are dynamic, which is why a static analysis alone is not enough to translate MAG.

Interestingly, we have noticed that every high level programming language which are used for game development are capable of reflection and although game engines lift the code, strip and obfuscate the binaries they have to retain these properties. One of these properties are, the namespace and name of a class, for example in Java there is `object.getClass().getName()`, in C# there is `object.GetType().Name` etc. CheatFighter injects a shared library into the game with a simple function that utilizes these APIs or their implementation in low level language to resolve an address or pointer, and then uses Frida to call that function and resolve the nodes in the MAG. This way the other compiled executables are untouched and we can use Frida to load this after everything else has been loaded into the memory keeping the memory layout intact.

Furthermore, this method is independent of the specific release of a game as the reflection property is usually under the core object which rarely changes. This whole process has been shown in ❸ of Figure 3, where each leaf node of the MAG points to a field but the immediate parent points to a data structure. We can get the offset of the field into the object from MAG but to get the name of the class we need to upcast it to the base or super object containing reflection properties such as classname. These two information together can help in determining the target class and the field of the cheat or the particular leaf node. As a cheat can have multiple leaf nodes it can also have multiple target class and fields. Figure 6 shows the translated MAG for an open source game that we have chosen for our end-to-end case study. As we can see, the leaf node and its immediate parent tells us that this particular MAG targets the class `NetworkedWeapon` and the field `ammoInClips`.

3.4 Selective Client Obfuscation

The end result of CheatFighter is a cheat resistant version of the game client, achieved via selective code obfuscation. As mentioned in §2.4, our solution must achieve this across multiple game engines, without heavily modifying the current development pipeline, which would delay industry adoption. Thus, we propose a cross-engine, compiler-independent solution that requires zero intervention from the developers for selective and targeted obfuscation of the client. CheatFighter achieves this by working on the source code level and by relying on the compiler's ability to generate ASTs that can be modified.

For proof-of-concept, we used the ROSLYN [38] compiler to generate the AST for the user-developed source code. ROSLYN is a .NET compiler that offers C# and Visual Basic languages with rich code analysis APIs. We have selected this because popular game engines use C#, even for Android game development where the code is transformed by the game engine. However, other tools such as clang can be employed to achieve this for other languages like C, C++, and so on. Once the AST is generated, CheatFighter locates the targeted data structure and fields from last step in the AST. Once identified, CheatFighter changes the position of the field in the data structure simply by swapping two nodes of the AST. Furthermore, it adds some padding fields to change the signature of the data structure to make it even harder for cheating tools to locate the data structure in memory by signature matching.

A brief overview of this process can be seen in ❹ of Figure 3, which starts by generating the AST and identifying the target based on the last step. In this case, the target class `a` and the target field `b` are marked by red. Once identified, CheatFighter changes the position of the field, thus changing the offset into the class and adds padding data (`d` in figure) to further change the signature of the data structure.

3.5 Other Platforms

Due to the difference in architecture, development process, and final execution environment, both games and cheats are different on Windows. However, we have found that memory modifying cheats for Windows games follow the same principles and the cheat intelligence can still be modeled into a MAG. The extraction process is very similar to the design mentioned above, with slight modifications to accommodate the differences.

Cheat Format. The resulting cheat for Android, which is based on Linux, can be an executable in ELF format (embedded in a larger APK), while for Windows it has to be in PE format. However, this detail is irrelevant for the operation of CheatFighter, as it works on the lifted IR rather than the binary itself. This abstraction enables us to use a high-level signature that is independent of the architecture or platform specifics.

Cheat Analysis Tool. Although `Ghidra` can lift the binary to PCode, it struggles with handling Windows-based PE files, so we have opted for the state-of-the-art tool IDA Pro for analyzing the Windows cheats. Like `Ghidra`, IDA also lifts the binary to an IR and offers a set of APIs to parse, transform and analyze the binary.

Memory Mapping. Lastly, we have seen that memory mapping is not as easily accessible in Windows and thus the head node of MAG on Windows often starts with the base address of the game rather than some referencing name. This reduces the complexity as the responsibility of the first component becomes half. Second, to read the memory of another process, windows cheats directly employ debug APIs, such as `KERNEL32` APIs, instead of memory file as in Linux or Android.

However, apart from this, the rest of the CheatFighter pipeline for extracting the MAG and applying selective code randomization remains the same. This demonstrates the generality and platform independence of our MAG encoding.

4 EVALUATION

We have implemented CheatFighter on top of Ghidra 10.0 with 4,540 lines of our own Java code to analyze the cheat binaries. We have used Frida 15.1.16 for dynamic analysis and Android NDK r23 LTS to compile the MAG translation shared library. We have used the latest Roslyn compiler to generate and modify the AST for client obfuscation. In this section, we present the evaluation results. We first describe our experiment setup, including how we collected the cheats, in §4.1, and then present the effectiveness and efficiency of our tool in §4.2 and §4.3, respectively. Finally, we present additional findings in §4.4.

4.1 Experiment Setup

Just like malware, it is trivial to collect a few cheats, but it is challenging to collect dozens of cheats due to the lack of a centralized distribution point. We applied two strategies in collection: 1) *Searching*. We use Google Search to locate possible cheats or groups that sharing cheats such as Telegram channels by keywords such as "PUBG wall hack". However, we found that only few cheats are real and the rest are indeed malware, and only few Telegram channels are still active. It costs us significant time and efforts to identify the valid information from the massive results; 2) *Expansion*. We found that in the cheats we have collected, they often contain vendor information or group information such as vendor website. Meanwhile, cheats distributors would send advertisements on the Telegram channel to attract users to join other Telegram channels, such as cheat sharing channels. As such, we enlarge our dataset by following those information and collect more cheats.

As a result, we have collected 86 unique cheats from multiple sources. In particular, we collected 34 cheats from Telegram channels where channel members would share some cheats and 41 cheats from the Kuaimao forum in which there is a sub-forum where users upload purchased or self-made cheats. An additional 10 cheats were

Table 3: The targeted victim games and their cheats (*#Installs based on US Play Store Data, A : Android,W: Windows)

Victim Game	Platform	Release	#Installs	Engine	#Cheats	#Binary
CrossFire Mobile	Android	12-03-2015	-	Unity	1	1
Arena 5v5	Android	11-30-2016	10M+	Unity	29	55
PUBG Mobile	Android	03-23-2017	500M+	UE4	35	74
COD Mobile	Android	10-01-2019	100M+	Unity	4	24
Royal Match	Android	02-25-2021	10M+	Unity	1	1
LOL	Android	10-27-2021	-	Unity	1	14
PUBG New State	Android	11-11-2021	10M+	UE4	2	2
Sausage Man	Android	04-29-2022	10M+	Unity	2	18
Assault Cube	Windows	04-01-2022	-	CUBE	1	1
Bard's Tale	Windows	06-17-2005	-	Dark Alliance	1	1
Super Tux	Windows	12-22-2021	-	SuperTux	1	1
COD MW3	Windows	11-08-2011	-	IW	1	1

collected from various sources, such as UnknownCheat website [8] and Discord. We collected the Windows cheat by searching for usage of a popular cheating library for windows. To get a better understanding we have even created a cheat for a well maintained online multiplayer open source game [5] (1,400+ stars on Github) following current cheat developing practices. San Andreas is a popular game by Rockstar, which was later ported to Android using the game engine Unity, thus the name San Andreas Unity.

Selection Criteria. Since the sources from which we collect cheats are used to share a variety of cheats, we have to filter to identify memory modifying ones. To this end, we have created two polices to identify relevant cheats:

(1) **Cheat Format.** Cheats can be developed using scripting languages such as Lua. Our targeted cheats are those compiled into native binaries. We identify such cheats by inspecting whether the cheat contains ELF binaries for Android or Windows.
(2) **Cheat Type.** We focus on memory modifying cheats, which requires accessing "`/proc/$pid/maps`" (or its aliases) or debug APIs. We filter the cheats that do not contain references to these strings or APIs.

With these policies, we eventually collected 86 cheats that target 13 popular online multiplayer games. A high-level overview of the cheats collected and the targeted victim games is presented in Table 3, sorted by the release date of the game. The number of installs for games available on the US Google Play store show that the most popular game, `PUBG Mobile`, has more than 500M+ downloads. Furthermore, the Android games are built on two of the biggest game engines in the market: Unity and Unreal Engine 4 (**UE4**). Finally, it shows the number of binaries found in those cheats, and it appears that a cheat may have multiple binaries for various objectives.

Environment Setup. The experiment was carried out on a single Intel Core i7-8700 3.2GHz machine with 16GB DDR4-2666MHz memory running Ubuntu 20.04. 4 instances of Ghidra were run concurrently in headless mode to analyze the cheat binaries and extract MAG.

4.2 Effectiveness

Quantifying False Positives & False Negatives. By applying CheatFighter on the 86 cheats, we have extracted MAG for 80 of them. During the analysis, 193 binaries have been identified from the cheats and all of them contain references to the string "`/proc/$pid/maps`" or debug APIs. From the binaries, CheatFighter identified 5,399 functions of interest and analyzed 12,351 instructions. As a result, CheatFighter produced 31 unique MAG for the 13 games. With these results, we first need to quantify the false positives and false negatives of CheatFighter before diving deeper into the results. We define a false positive (FP) as the event of identifying a MAG that has not been used to access or modify the memory, and a false negative (FN) as the event of CheatFighter failing to extract the MAG even though it is present in the cheat.

FP Analysis. To check the FP, we randomly selected a cheat from each game and compared the MAG with our manual finding. Note that for the cheats selected for the validation, we have made no assumption about their particular implementation, and the manual analysis was done on the decompiled source code produced by Ghidra. Encouragingly, we have found that all the memory access graphs extracted by CheatFighter aligned with our manual findings without any false positives.

FN Analysis. To our surprise, CheatFighter did not produce a MAG for 6 cheats (with an FN rate of 6.97%). We inspected these cheats and found out the root cause is that the offset used to traverse memory comes from file inputs, not values hardcoded in the program code. Particularly, we noticed that these cheats read the offset from a file (e.g., the temporary file `/sdcard/Android/.me`) into an array and then use them as offset at different places of the pointer chain. Since CheatFighter does not dynamically execute the cheat, and does not currently model filesystem accesses, it will not yield a complete MAG in such cases. We discuss our prototype's limitations and planned future work in §5.

Detailed Results. Table 4 presents the result for memory access graphs extracted by CheatFighter, and this result is categorized by the game they are targeting and is divided into *five major sections* starting with the input, analysis both in forward and backward (back slicing for pointer analysis) and finally the result along with the type of memory access done by the cheats. To avoid analyzing any duplicated cheats, CheatFighter uses the MD5 hash of the cheat to index each of them. However, multiple cheats, though they have different MD5s, they produce the same memory access graphs, indicating they shared the same intelligence. Therefore, we do not report the detailed results for these cheats and instead use the averaged result in Table 4 to present them with a count number, and a more detailed version of their result presented in Table 6 in Appendix. The input shows the number of binaries found in a particular cheat and their size in KB. For cheats with multiple binaries, the combined size is presented. Details regarding the analysis are represented in terms of functions, instructions, and PCode analyzed by our tool. Finally, the output is presented by the number of times a MAG is observed in different cheats, height, number of branches and edges in the MAG produced. The output section also contains the number of bases or referencing name used by a cheat along with their name. The detailed results are presented below, categorized by the component related as explained in §3.

Table 4: MAG evaluation result. Note that for the rows without MD5 values, we present the total number of cheats in that row, since these cheats have the same MAG.

			Input		Forward Analysis			Backward Analysis			Output					Memory Access	
Game	Platform	Cheat MD5	#Bin	∑Size	#Func	#Ins	#PCode	#Func	#Ins	#PCode	Height	#Branch	#Edges	#Base	Bases	Read	Write
CFM	**Android**	4e349c25d1c5e303f73b9fa8b94934dd	1	652	60	3,754	15,215	60	61,009	212,449	1	3	3	2	[TS][U]	✓	✗
Arena 5v5	**Android**	0a45409737c036f9d59c5feb427ba9c5	1	12	4	11	42	4	69	220	1	1	1	1	[GC]	✓	✗
		2e8268d32dc22c31dd8579bca6b7f7d7	4	48	16	44	168	16	276	880	1	1	1	2	[GC][U]	✓	✗
		* (10)	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
		11a0894dd4bcc57fac657dc244e597a8	1	1,039	3	47	219	3	868	2,718	7	14	46	1	[GC]	✓	✗
		c632aeaaecbe67487f0bf6f69416cb38	1	21	7	319	1,459	7	3,356	8,521	8	16	55	2	[GC][U]	✓	✗
		* (3)	1	79	3	103	501	3	558	1,393	8	16	55	3	[GC][U][IL]	✓	✗
		* (4)	1	83	4	171	841	4	773	2,047	8	20	66	3	[GC][U][IL]	✓	✗
		* (4)	3	129	10	315	1,591	10	1,235	3,334	8	22	72	2	[GC][U]	✓	✗
		* (2)	2	139	9	382	1,966	9	1,314	3,499	8	26	89	2	[GC][IL]	✓	✗
		76bab2ee423c05c1b6f10abc52653683	1	358	16	412	1,182	16	4,663	13,299	10	14	122	2	[GC][IL]	✓	✓
		04caab0a8f0b10d7750ae1d424034a7b	3	41	11	175	837	11	9,727	29,105	10	22	74	3	[GC][U][IL]	✓	✗
PUBG	**Android**	397446459fe284a2c10f676b57c03982	1	13	3	6	22	3	46	152	1	1	1	1	[UE4]	✓	✓
		* (19)	1	13	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
		* (3)	1	13	4	11	42	4	69	220	1	1	1	2	[UE4][GC]	✓	✗
		* (8)	1	14	4	16	71	4	73	229	1	2	2	1	[UE4]	✓	✗
		* (3)	14	184	55	198	865	55	983	3,104	1	5	5	2	[UE4][TS]	✓	✓
		9a2cec9ac23cc6b9713d983d202a04ed	1	13	4	11	51	4	58	190	5	1	5	1	[UE4]	✓	✓
COD	**Android**	* (3)	1	12	4	11	42	4	69	220	1	1	1	1	[IL]	✓	✗
		ac97f45290f238e5346f0ef5ae839cb9	21	269	83	264	1,032	83	1,568	4,945	1	3	3	3	[IL][GC][U]	✓	✓
Royal Match	**Android**	c8b4767dc7a0b57ce608173cbc7e6b15	1	8	4	31	115	4	2020	7445	6	1	7	1	[IL]	✓	✓
LOL	**Android**	6cdee600b5085c0c1d27c2a4d1654869	14	202	59	195	851	59	990	3197	1	12	12	4	[GC][TS][U][NPP]	✓	✓
Sausage Man	**Android**	38fa5a9ba3a271ec9e2ad0724eae24d9	4	68	18	104	353	18	792	2546	4	5	5	2	[IL][U]	✓	✗
		217ac1c9109a9e0103d364a4356dbd40	14	527	56	201	883	56	1117	3391	10	15	70	2	[IL][U]	✓	✓
PUBG2	**Android**	bf111d5d095f9dc0d597cf0c93af7791	1	13	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
		214d2b41ba49a3773c66befc0e1a4e4c	1	12	4	11	42	4	69	220	1	1	1	1	[GC]	✓	✗
SA Unity★	**Android**	-	1	8	4	31	117	4	2121	7638	8	1	8	1	[IL]	✓	✓
Assault Cube	**Windows**	5c0d8bfbb3589032f846cebb699993e1	1	23	1792	1969	-	535	532	-	1	7	7	1	[S]	✓	✓
Bard's Tale	**Windows**	5dc6952102781bc2d8970d62f5d22a01	1	18	1392	1520	-	212	206	-	1	10	10	1	[S]	✓	✓
Super Tux	**Windows**	42b9cafa7a6153d00fe2654ee01387e0	1	15	690	771	-	270	264	-	1	6	6	1	[S]	✗	✓
COD MW3	**Windows**	3a2d4279b71d30b9d29887a44335375b	1	411	1066	1172	-	733	788	-	1	9	9	1	[S]	✗	✓

[GC]: libGameCore, [IL]: libil2cpp, [U]: libunity, [UE4]: libUE4, [TS]: libtersafe, [NPP]: libNssPhysicsPlugin, [S]: Self ★Custom cheat for open source game

Head Node Identification. The focus of the head node identification is to find the referencing name and the base address, but depending on the referencing name that a cheat uses, we can infer where the logic of the code lives. That is depending on the engine used, and the main logic of a game can often be found in the shared object(s) or libraries used in a game and these also contain the location of the critical variables that a cheat developer wants to access. Table 4 shows the referencing name used by each cheat. From the table it is apparent that for all `Unity` games, `libil2cpp.so` works as a preferable base address, i.e., most critical data can be found in the said shared object.

Moreover, we can also see that a library can be used to attack a number of games, such as `libGameCore`, which is a common library used in game development. We have seen this library used as a base for 5 out of 13 games we have analyzed. From the list of referencing name the cheat targets we can also identify libraries that need to be protected to invalidate any memory-modifying cheat. To be more specific, take Figure 2 for example, by identifying the libraries used for base address, developers can compile them with different orderings for the data structure or different compiler optimizations to invalidate all cheats targeting these libraries on a high level.

Children Node Discovery. We have chosen 3 distinct properties to explain the complexity of a MAG: height, number of branches, and edges. For example, from Table 4, we can see that more memory accesses are performed for targeting `Arena 5v5` than other games, resulting in more nodes and complex networking in the MAG, which is evident in Figure 2. This means that creating a working cheat for `Arena 5v5` requires more skill and time. Furthermore, complicated graphs are easier to break via randomization since there are more candidate edges to modify in future versions of the game client. In addition, it can be observed that with the presence of multiple cheats for the same game, we can speculate how frequently the data structures or the memory layout changes due to the update in client version. `PUBG` is more prone to frequent updates than other games in our dataset, leading to it having many cheats with different MAG.

Case Studies. We present a case study for a MAG that is contained in 19 cheats for `PUBG Mobile`. The decompiled code in Figure 7 shows the simplest MAG with only one offset. The associated data structures for these addresses can be used by anti-cheating software, as robust signatures [19, 20, 37] to detect cheaters currently using these cheats, thus defending against a line of cheaters as well. As stated earlier, it is common for different cheats to use the same MAG which is possible due to the knowledge sharing among cheat developers via dedicated websites [8]. Thus, by using one signature (MAG), anti-cheating tools can block multiple polymorphic cheats.

Furthermore, it can be observed from Figure 2 that for any sub-path the first offset is significantly greater than the others. Across

```c
int main(void) {
    char *package = "com.tencent.tmgp.pubgmhd";
    int pid = getPID(package);
    char *library_name = "libUE4.so";
    long base_address = get_module_base(pid,library_name);
    long node_1 = base_address + 0x41b6ae0;
    float new_value = 75.0;
    WriteAddress_FLOAT(package,node_1,new_value);
    puts(&DAT_00101818);
    return 0;
}
```

Figure 7: Case Study for PUBG

different versions of a game, a change in this first offset is inevitable, but a portion of the MAG can still be used in subsequent cheats. Thus, by breaking the smaller sub-chains that have existed across different cheats, developers can further stop the usage of similar attacks. For instance, considering the two sub-paths for the game `Arena 5v5`: *1)* ((((base_address(libil2cpp.so) + 0x9759f44)* + 0x50)* +0x08)* + 0xa4)* and *2)* ((((base_address(libil2cpp.so) + 0x6873f9c)* + 0x50)* +0x08)* + 0xa4)*, these two sub-paths differ from each other by only one node and the sub-chain has been found in 16 binaries from 3 different cheats for the game `Arena 5v5`. Developers can take advantage of this knowledge to defend against every cheat using this sub-path only.

4.3 Efficiency

To illustrate the efficiency of CheatFighter, we present Figure 8 that shows the time taken by both Ghidra and CheatFighter to analyze a given binary with respect to the different sizes of the binary. As mentioned earlier, CheatFighter is implemented as a post-analysis Ghidra Script, i.e., it works after Ghidra has finished its analysis. Therefore, it can be seen that for most binaries the time by CheatFighter is significantly lower, which is expected given the heavy analysis Ghidra executes. However, we can also see that the size of the binary does not affect the analysis time linearly. That is, for binary sized from 12KB to 100KB, the time taken by Ghidra remains the same, and this is also true for CheatFighter. Surprisingly, when the size reaches 350KB+ the time suddenly jumps to almost more than 6 times. We also observed that the time taken by CheatFighter also increased with this case. Upon further investigation it was discovered that the time is directly correlated with the number of pointers analyzed by CheatFighter in the binary.

4.4 Other Findings

Effect of Engine. Games built with the same engine often share a similar implementation. This behavior can be extended to the memory structure of the game and the cheats that target it. Table 5 shows the difference in the analysis for the engines in terms of input (summation) and output (average). From Table 5 it is clear that although having the same or even more cheats of UE4, the number of binaries, size, and instructions are all greater for Unity. Similarly, the output, i.e., memory access graphs produced by cheats for Unity are much more complicated with taller and wider graphs, which also take longer to analyze.

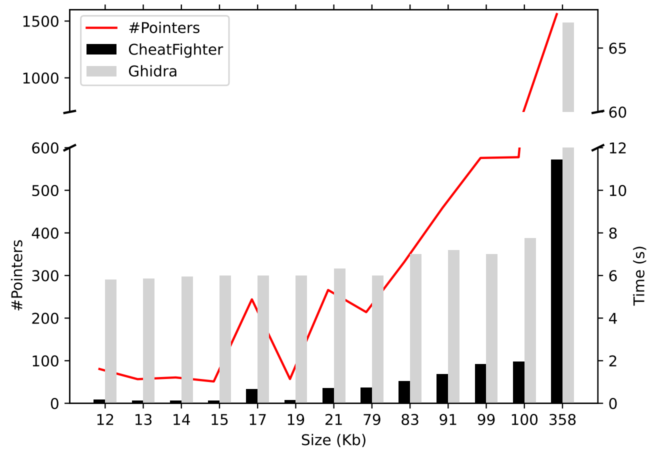

Figure 8: Efficiency of CheatFighter

Table 5: Engine Impact on Our Analysis

		Engine	
		Unity	**UE4**
∑Input	**#Cheats**	38.00	37.00
	#Binaries	153.00	65.00
	Size	5,451.00	839.00
	#Ins	116,586.00	5,697.00
$\overline{Output}$	**Common Base**	libIl2cpp.so	libUE4.so
	Height	14.39	1.55
	Branch	28	7.82
	#Edges	78.09	8.55
	Time(s)	61.98	25.88

5 LIMITATIONS & FUTURE WORK

While our prototype demonstrates the feasibility of a rapid automated response to widespread cheating, we do not claim that it currently handles all conceivable program behaviors. Specifically, while it was infrequent in our evaluation dataset, it is possible for a cheat program to read offsets from a configuration file rather than directly encoding them into the binary. Such cases can be handled by modeling file system accesses, which we leave for future work.

It is also possible for a cheat program to linearly scan memory for a specific signature as opposed to traversing objects, however this is of limited utility to the cheater because false positive signature matches can result in the wrong values being modified, crashing the game. We believe this is why we did not witness any cases of linear scanning in our dataset. Even if such cases arise, they will be thwarted by the same randomization used to break the pointer chains, which involves adding padding data to data structures.

We did not evaluate our prototype's robustness to obfuscation and circumvention, since we could not find any such real-world examples. If a program were to obfuscate a string like "`/proc/$pid/maps`", it would have to eventually decode it into its plain text value in order to make a valid system call, at which point our system's analysis would recover it. CheatFighter can also incorporate ideas from prior work to handle obfuscation [27, 31, 32, 43, 52]. Circumvention is difficult for the cheat program assuming that the data targeted by the cheater (and identified by CheatFighter) is randomized differently in each game client. Even if it is not and a cheater updates their cheat program to account for the new object layouts, since CheatFighter is fully automated and completes its analysis in under 1 minute in most cases, repeating the randomization is trivially inexpensive.

Finally, we focus on a prevalent and hard-to-defend type of cheat, as discussed in §2.1. However, there are several cheats that depend on the architecture, platform, or development process, which CheatFighter does not address. For example, one way to cheat is to use modified DLL injection for Windows-based games and Shared Object injection for Linux-based games. Moreover, games developed with Mono are also vulnerable to DLL modification attacks. However, there are already several defenses that can detect such injection-based tampering of the game client. In contrast, the type of cheat we have targeted in this work operates from outside the game by leveraging the capability of the operating system as a whole. Such prior tampering detection techniques cannot handle the scenario our work addresses.

6 RELATED WORK

Anti-Cheating. There have been numerous approaches proposed to defend against game cheating [36, 51], and these include from code obfuscation [21, 42], process monitoring [24, 28, 41], network traffic analysis [17, 33], server side verification [13], private set intersection protocols [15], trusted execution environment [12, 40], to recently trusted execution environment [12, 40] and more proactive vulnerability discovery [10, 56]. Compared to these works, we explore new directions of automatically analyzing cheat binaries to uncover how a cheat would attack the victim games, and then use the extracted intelligence to defend against the cheating reactively.

Memory Access Chain Analysis. Uncovering how a program variable is accessed (e.g., through pointer dereference) is extremely valuable in security applications such as kernel rootkit detection [23] and cross-version memory forensics [22]. In particular, FpCK [23] developed a dynamic analysis based approach to uncover the pointer access chain, and applied it to detect kernel function pointer corruptions. Origin [22] introduced the notation of offset-revealing instruction and also used dynamic analysis to uncover the memory access chain of user level programs. Lastly, SigPath [48] uses memory snapshots to build a path to the variable of interest, the same method as used by cheat developer. Compared to these dynamic analysis based approaches, CheatFighter explores the static binary analysis for memory access chain recovery, and demonstrated in novel game anti-cheating applications.

Binary Analysis. CheatFighter has integrated many of the existing fundamental program analysis including backward slicing [50], data dependence analysis [34] (or taint analysis [39, 55]), and value set analysis [11], while developing our own API-aware, context-sensitive, memory graph extraction algorithm from cheat binaries. Numerous works have also been done that focus on binary analysis (e.g., improving binary analysis techniques or applying them to solve security problems such as malware analysis and vulnerability discovery) [44]. In this work, we extend the binary analysis to a new game cheat target.

7 CONCLUSION

We have presented CheatFighter, a new automated defense against widespread cheating in video games that makes use of cheat intelligence extracted directly from widely distributed and sold cheat programs. Specifically, we propose to extract the pointer chains and references from cheat binaries to guide selective data structure randomization to invalidate the cheat. We propose MAG as the encoding representing the unified cheat intelligence scattered in cheats to guide our defense. We have developed a prototype of CheatFighter and tested with 86 cheats targeting 13 popular games, and shown that CheatFighter is capable of automatically extracting MAG and using it to subsequently harden game clients via selective randomization.

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to the anonymous reviewers for their constructive and insightful comments, which helped us improve the quality and clarity of this paper. We also thank the shepherd, for their valuable guidance and support throughout the revision process. This research was supported in part by ARO award W911NF2110081, DARPA award N6600120C4020, and NSA award H98230-22-1-0333,

REFERENCES

[1] [n. d.]. The business impact of video game cheaters and pirates. https://venturebeat.com/2021/06/01/the-business-impact-of-video-game-cheaters-and-pirates-and-how-to-fight-back-vb-live/. (Accessed on 09/25/2021).
[2] [n. d.]. Il2CppDumper. ([n. d.]). https://github.com/Perfare/Il2CppDumper.
[3] [n. d.]. Newzoo: Game market will hit $200B in 2024 | VentureBeat. https://venturebeat.com/2021/07/04/newzoo-game-market-will-hit-200b-in-2024/. (Accessed on 09/25/2021).
[4] [n. d.]. Parallel Space Virtual Space: No Root for GameGuardian. https://www.thedroidmod.com/2019/07/gameguardian-parallel-space.html. (Accessed on 01/25/2021).
[5] [n. d.]. San Andreas Unity Edition. ([n. d.]). https://github.com/GTA-ASM/SanAndreasUnity.
[6] [n. d.]. Securing Unity Games with DexGuard and iXGuard. https://www.guardsquare.com/blog/securing-unity-games-dexguard-and-ixguard-how-it-works. (Accessed on 09/25/2021).
[7] [n. d.]. Security-Enhanced Linux in Android. https://source.android.com/security/selinux. (Accessed on 09/25/2021).
[8] [n. d.]. UnKnoWnCheaTs. https://www.unknowncheats.me/forum/index.php. (Accessed on 10/01/2022).
[9] 2021. Game Guardian. https://gameguardian.net/
[10] Luigi Auriemma1 and Donato Ferrante. 2023. GAME ENGINES: A 0-DAY'S TALE. In *NoSuchCon'23*. Paris, France. https://dl.packetstormsecurity.net/papers/general/ReVuln_Game_Engines_0days_tale.pdf
[11] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–23.
[12] Erick Bauman and Zhiqiang Lin. 2016. A Case for Protecting Computer Games With SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*. Trento, Italy.
[13] Darrell Bethea, Robert A Cochran, and Michael K Reiter. 2008. Server-side verification of client behavior in online games. *ACM Transactions on Information and System Security (TISSEC)* 14, 4 (2008), 1–27.
[14] Reinhard Blaukovitsch. 2020. YOU MIGHT BE SURPRISED BY WHAT YOUR MOBILE GAMERS LOOK LIKE. https://blog.irdeto.com/video-gaming/you-might-be-surprised-by-what-your-mobile-gamers-look-like/

[15] Elie Bursztein, Mike Hamburg, Jocelyn Lagarenne, and Dan Boneh. 2011. Openconflict: Preventing real time map hacks in online games. In *2011 IEEE Symposium on Security and Privacy*. IEEE, 506–520.

[16] N. Cano. 2019. Cheat Engine. https://www.cheatengine.org/

[17] Kuan-Ta Chen, Jhih-Wei Jiang, Polly Huang, Hao-Hua Chu, Chin-Laung Lei, and Wen-Chin Chen. 2008. Identifying MMORPG bots: A traffic analysis approach. *EURASIP Journal on Advances in Signal Processing* 2009 (2008), 1–22.

[18] Nicholas Cole, Sushil J Louis, and Chris Miles. 2004. Using a genetic algorithm to tune first-person shooter bots. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, Vol. 1. IEEE, 139–145.

[19] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T King. 2008. Digging for Data Structures.. In *OSDI*, Vol. 8. 255–266.

[20] Brendan Dolan-Gavitt, Abhinav Srivastava, Patrick Traynor, and Jonathon Giffin. 2009. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security*. 566–577.

[21] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. 2011. Multi-stage binary code obfuscation using improved virtual machine. In *International Conference on Information Security*. Springer, 168–181.

[22] Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. 2016. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 11–22.

[23] Yangchun Fu, Zhiqiang Lin, and David Brumley. 2015. Automatically deriving pointer reference expressions from binary code for memory dump analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 614–624.

[24] GameGuard 2012. http://gameguard.nprotect.com/en/index.html. Retrieved 5/4/2018.

[25] GameKiller. 2021. GameKiller. http://game-killer.com/

[26] Nelson Granados. 2018. Report: Cheating Is Becoming A Big Problem In Online Gaming. https://www.forbes.com/sites/nelsongranados/2018/04/30/report-cheating-is-becoming-a-big-problem-in-online-gaming/?sh=702cc8597663

[27] Yoann Guillot and Alexandre Gazet. 2010. Automatic binary deobfuscation. *Journal in computer virology* 6, 3 (2010), 261–276.

[28] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. 2010. Accountable Virtual Machines.. In *OSDI*. 119–134.

[29] Munawar Hafiz and Ming Fang. 2016. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering* 21, 5 (2016), 1920–1959.

[30] Kiwon Hong, Youngjun Kim, Hyungoo Choi, and Jinwoo Park. 2017. SDN-assisted slow HTTP DDoS attack defense method. *IEEE Communications Letters* 22, 4 (2017), 688–691.

[31] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. 2017. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security* (Reggio Calabria, Italy) *(ARES '17)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. https://doi.org/10.1145/3098954.3098995

[32] Zeliang Kan, Haoyu Wang, Lei Wu, Yao Guo, and Daniel Xiapu Luo. 2019. Automated deobfuscation of Android native binary code. *arXiv preprint arXiv:1907.06828* (2019).

[33] Ah Reum Kang, Jiyoung Woo, Juyong Park, and Huy Kang Kim. 2013. Online game bot detection based on party-play log analysis. *Computers & Mathematics with Applications* 65, 9 (2013), 1384–1395.

[34] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2017. *Data flow analysis: theory and practice.* CRC Press.

[35] Junbaek Ki, Jung Hee Cheon, Jeong-Uk Kang, and Dogyun Kim. 2004. Taxonomy of online game security. *The Electronic Library* (2004).

[36] Samuli Johannes Lehtonen et al. 2020. Comparative Study of Anti-cheat Methods in Video Games. (2020).

[37] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. 2011. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures.. In *Ndss*.

[38] Microsoft and Contributors. 2023. Roslyn: The .NET Compiler Platform ("Roslyn") provides open-source C# and Visual Basic compilers with rich code analysis APIs. https://github.com/dotnet/roslyn Accessed: 2023-07-03.

[39] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.. In *NDSS*, Vol. 5. Citeseer, 3–4.

[40] Seonghyun Park, Adil Ahmad, and Byoungyoung Lee. 2020. BlackMirror: Preventing Wallhacks in 3D Online FPS Games. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 987–1000.

[41] Punkbuster 2018. http://www.evenbalance.com/. Retrieved 5/4/2018.

[42] Rolf Rolles. 2009. Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies* (Montreal, Canada) *(WOOT'09)*. USENIX Association, USA, 1.

[43] Hassen Saıdi, Phillip Porras, and Vinod Yegneswaran. 2010. Experiences in malware binary deobfuscation. *Virus Bulletin* (2010).

[44] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[45] Deris Stiawan, Mohammad Yazid Bin Idris, Abdul Hanan Abdullah, Mohammed AlQurashi, and Rahmat Budiarto. 2016. Penetration Testing and Mitigation of Vulnerabilities Windows Server. *Int. J. Netw. Secur.* 18, 3 (2016), 501–513.

[46] I. Melih Tas, Basak Gencer Unsalver, and Selcuk Baktir. 2020. A Novel SIP Based Distributed Reflection Denial-of-Service Attack and an Effective Defense Mechanism. *IEEE Access* 8 (2020), 112574–112584. https://doi.org/10.1109/ACCESS.2020.3001688

[47] Ruck Thawonmas, Yoshitaka Kashifuji, and Kuan-Ta Chen. 2008. Detection of MMORPG bots based on behavior analysis. In *Proceedings of the 2008 International Conference on Advances in Computer Entertainment Technology*. 91–94.

[48] David Urbina, Yufei Gu, Juan Caballero, and Zhiqiang Lin. 2014. SigPath: A Memory Graph Based Approach for Program Data Introspection and Modification. In *Computer Security - ESORICS 2014*, Mirosław Kutyłowski and Jaideep Vaidya (Eds.). Springer International Publishing, Cham, 237–256.

[49] Steven Daniel Webb and Sieteng Soh. 2007. Cheating in Networked Computer Games: A Review. In *Proceedings of the 2nd International Conference on Digital Interactive Media in Entertainment and Arts* (Perth, Australia) *(DIMEA '07)*. Association for Computing Machinery, New York, NY, USA, 105–112. https://doi.org/10.1145/1306813.1306839

[50] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[51] Jiyoung Woo and Huy Kang Kim. 2012. Survey and Research Direction on Online Game Security. In *Proceedings of the Workshop at SIGGRAPH Asia* (Singapore, Singapore) *(WASA '12)*. Association for Computing Machinery, New York, NY, USA, 19–25. https://doi.org/10.1145/2425296.2425300

[52] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *2015 IEEE Symposium on Security and Privacy*. 674–691. https://doi.org/10.1109/SP.2015.47

[53] Jeff Yan and Brian Randell. 2005. A Systematic Classification Of Cheating In Online Games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. ACM, 1–9.

[54] Jeff Yan and Brian Randell. 2005. A systematic classification of cheating in online games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. 1–9.

[55] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*. 116–127.

[56] Chaoshun Zuo and Zhiqiang Lin. 2022. Playing Without Paying: Detecting Vulnerable Payment Verification in Native Binaries of Mobile Games. In *31th USENIX Security Symposium (USENIX Security 22)*.

		Input		Forward Analysis			Backward Analysis			Output					Memory Access	
Game	**Cheat MD5**	**#Bin**	**∑Size**	**#Func**	**#Ins**	**#PCode**	**#Func**	**#Ins**	**#PCode**	**Height**	**#Branch**	**#Edges**	**#Base**	**Bases**	**Read**	**Write**
Arena 5v5	ed20772d1d946397668f246635949064	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	f53dc3d3e78c72a3a1637b37d3c85d43	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	c90b7345fc08e43cc27b59355f8b1baf	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	0cdb69c9c20ff78447778210d96337da	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	078b1ecde9d4a9914bc736bd9d85445c	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	9fdfdf42070981a6dd60b694f53c5b02	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	9423af216e97630319390ecd3dc121bf	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	c0e39becdaa1a5d1667989619ab06243	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	c84136a77a7f712f7d583c4b29958bf6	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	6519935962e97fec12693f4dfaf428e1	2	24	6	74	330	6	768	2,358	6	8	25	1	[GC]	✓	✗
	7b82d33ba38aa5542d5f5f50310bd48a	1	79	3	103	501	3	558	1,393	8	16	55	3	[GC][U][IL]	✓	✗
	4d06250057e4c0d6ec30980749230495	1	79	3	103	501	3	558	1,393	8	16	55	3	[GC][U][IL]	✓	✗
	4ea326575370827b37d63ccf71b7ad08	1	79	3	103	501	3	558	1,393	8	16	55	3	[GC][U][IL]	✓	✗
	dac338cc90b680e0d271d43229605e0f	1	83	4	171	841	4	773	2,047	8	20	66	3	[GC][U][IL]	✓	✗
	e576bc6a6bb40c2121ed20b87adda03a	1	83	4	171	841	4	773	2,047	8	20	66	3	[GC][U][IL]	✓	✗
	16e88fb8be1a9d90614e9cb996f9d3ea	2	103	8	252	1,273	8	875	2,358	8	22	72	2	[GC][U]	✓	✗
	2626ffae6cd0cc30d5bf6c2cfb2b87b8	2	103	8	252	1,273	8	875	2,358	8	22	72	2	[GC][U]	✓	✗
	e7259e23e293b117be24376ebf4b8f8c	2	103	8	252	1,273	8	875	2,358	8	22	72	2	[GC][U]	✓	✗
	c2794a5edd945a9c1dc6e7c5a74ae415	4	206	16	504	2,546	16	2,316	6,260	8	22	72	2	[GC][U]	✓	✗
	f115e1590c73f651693f64fcae962302	1	100	4	297	1,539	4	888	2,332	8	26	89	2	[GC][IL]	✓	✗
	962af48bdbaa7bc666cf9deff44e982a	2	119	8	308	1,580	8	957	2,552	8	26	89	2	[GC][IL]	✓	✗
	70af7cd3fd33ca82cd2bfa12226828bf	2	112	8	308	1,581	8	957	2,552	8	26	89	2	[GC][IL]	✓	✗
	d08e521f1f14d69e3b4e1a248a32fa78	4	223	16	616	3,162	16	2,455	6,558	8	26	89	2	[GC][IL]	✓	✗
	fcbe24be5c57eead0c9b39cafb2e6943	1	14	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	ee19ba7ba66cf7346bc98c36dddd158c	1	15	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	d822844ed6687f37e922d9327b2c065c	1	13	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	d54b8e1f404342eb1d1f521738fa0ddf	1	12	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	bb683b8528b60419fcbc4226f989d709	1	15	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	ba188f4d6970f2598e44ca64a7bc2457	1	13	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	a96353cbffa388bd1291f08b18880f09	1	15	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	9d18e486f5386ebe8e8b9166d4e75317	1	15	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	9a923b4afef43a1ba9679e5eca828965	1	14	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	845d446534477915b365bbdb11cf255e	1	15	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	6c7a3b0f884fb20d85b926862f77e2aa	1	12	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	5848bf8853b2535d462337c45fb06ca5	1	14	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	48fe5e9600685e6120745183ef9b75ad	1	13	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	43544cbd685d87fd86c5a0c81f5a3f95	1	13	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	4113aa7a6da930fc6081e4067252ab30	1	12	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	28a1623a875d9bd6b869ec4a2f85fc56	1	12	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
PUBG	282e3fe905a74d9c8c1058038cdfeb95	1	12	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	261e1427c1e363f89cf6bfb21935e122	1	13	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	15278340f66803627b0aa815caf0589c	1	14	4	11	42	4	69	221	1	1	1	1	[UE4]	✓	✗
	89a0aaf71c03ebf9e078854487560075	1	13	4	11	42	4	69	220	1	1	1	2	[UE4][GC]	✓	✗
	045bada997809b5a2aed55962c7bfe20	1	13	4	11	42	4	69	220	1	1	1	2	[UE4][GC]	✓	✗
	03ce5d7e5cbd27fd8b9043b4481cfb95	1	13	4	11	42	4	69	220	1	1	1	2	[UE4][GC]	✓	✗
	eee3a703775c5e4fb70f9f44248981ad	1	14	4	16	71	4	73	229	1	2	2	1	[UE4]	✓	✗
	e7f871fa0c8aeb32d29d3d61b7a81fee	1	14	4	16	71	4	73	229	1	2	2	1	[UE4]	✓	✗
	e591ce32ab5a27e981081537959cdf91	1	14	4	16	71	4	73	229	1	2	2	1	[UE4]	✓	✗
	c9a3e73022d98c22429506fbb5a5a582	1	14	4	16	71	4	73	229	1	2	2	1	[UE4]	✓	✗
	7e95711effacd56b65f5e74888c28881	1	14	4	16	71	4	73	229	1	2	2	1	[UE4]	✓	✗
	58739954df87f5cf13cce7b1942987b4	1	14	4	16	71	4	73	229	1	2	2	1	[UE4]	✓	✗
	12487a0cca79077cbafc0f9bae7bbeeb	1	13	4	16	71	4	73	229	1	2	2	1	[UE4]	✓	✗
	05f46fbcbfa3fb795b547cef31ee4d51	1	14	4	16	71	4	73	229	1	2	2	1	[UE4]	✓	✗
	4c0c315fded1dd92e6604f5c7f9d1f11	14	184	55	198	865	55	983	3,104	1	5	5	2	[UE4][TS]	✓	✓
	33453e5c99317f487bb056693677c22a	14	184	55	198	865	55	983	3,104	1	5	5	2	[UE4][TS]	✓	✓
	13215660774577750c2774559d452d7e	14	184	55	198	865	55	983	3,104	1	5	5	2	[UE4][TS]	✓	✓
	6294fd69012c3b2d984798e64ae78828	1	12	4	11	42	4	69	220	1	1	1	1	[IL]	✓	✗
COD	f5a270e5d051cd6c56ffe321cebb27f1	1	12	4	11	42	4	69	220	1	1	1	1	[IL]	✓	✗
	4202525ebc54a901effc0e188903954e	1	12	4	11	42	4	69	220	1	1	1	1	[IL]	✓	✗

[GC]: libGameCore, [IL]: libil2cpp, [U]: libunity, [UE4]: libUE4, [TS]: libtersafe, [NPP]: libNssPhysicsPlugin

Table 6: Detailed Result