DOI: 10.15514/ISPRAS-2026-38(2)-4

C-code evolution lessons and real-time architecture patterns from the ioquake3 project

S.M. Staroletov, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
Polzunov Altai State Technical University,
46, prospect Lenina, Barnaul, Altai region, 656038, Russia.

Abstract. Modern compilers increasingly restrict code that could perform potentially dangerous actions. Experienced developers often look back with a smile on their old code, amazed at its shortcomings and how it ever functioned. On large-scale projects, such a look back can reveal all the flaws in the programming approach of past eras. This article examines the challenges of compiling and refactoring legacy code using Quake III (1999) as a case study. As a seminal title from the early era of 3D graphics accelerators, Quake III represents a kind of real-time system optimized for limited hardware, comparable to modern mobile or embedded devices. Its C open-sourced codebase, released as the ioquake3 project, now compilable for contemporary processor architectures, offers a unique approach to analyze compiler-driven code quality improvements. In this paper, we study the differences between the original code and the modern port code by clustering the changes in diffs between these projects. We also investigate how the game source code behaves under modern GCC with all warnings enabled. By categorizing compiler warnings and required fixes, we highlight patterns in unsafe practices and demonstrate necessary refactoring steps. Our findings aim to bridge historical coding practices with modern reliability standards, providing actionable insights for educators in C programming, maintainers of legacy systems, and developers interested in compiler-enforced code quality.

Keywords: code quality; diff analysis; code clustering; real-time-architecture; LLM use-case.

For citation: Staroletov S.M. C-code evolution lessons and real-time architecture patterns from the ioquake3 project. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 2, 2026., pp. 53-68. DOI: 10.15514/ISPRAS-2026-38(2)-4.

Уроки эволюции С-кода и шаблоны архитектуры реального времени на основе проекта ioquake3

С.М. Старолетов, ORCID: 0000-0001-5183-9736 <serg_soft@mail.ru>
АлтГТУ им. И.И. Ползунова,
Россия, 656038, Алтайский край, г. Барнаул, пр. Ленина, 46.

Аннотация. В современных компиляторах все сильнее действуют запреты на код, который может выполнять потенциально опасные действия. Опытные разработчики часто с улыбкой оглядываются на свой старый код, удивляясь его недостаткам и тому, как он вообще функционировал, а на масштабных проектах такой взгляд назад может показать все недостатки подхода к программированию в прошлые эпохи. В этой статье рассматриваются проблемы компиляции и рефакторинга legacy (унаследованного) кода на примере игрового движка Quake III (1999). Будучи знаковым проектом ранней эры 3D-графических ускорителей, игровой движок, по сути, представляет собой систему реального времени, оптимизированную для ограниченного аппаратного обеспечения – сопоставимого с современными мобильными или встроенными устройствами. Его открытая кодовая база на С, выпущенная в виде ioquake3 проекта, теперь компилируемая для всех современных архитектур, предоставляет уникальную возможность для анализа улучшения качества кода, исходя из предупреждений компилятора. В этой статье мы изучаем различия между оригинальным исходным кодом Quake III и кодом современного проекта ioquake3, кластеризуя изменения в diff-файле между этими проектами. Мы также исследуем, как исходный код игры ведет себя под современным GCC со всеми включенными предупреждениями. Категоризируя предупреждения компилятора и уже сделанные разработчиками необходимые исправления, мы выделяем шаблоны небезопасных практик и демонстрируем необходимые шаги по рефакторингу. Результаты работы призваны связать исторические практики программирования с современными стандартами надежности, предоставляя практические выводы для преподавателей программирования на С, разработчиков, сопровождающих унаследованные системы и разработчиков, интересующихся качеством кода.

Ключевые слова: качество кода; анализ изменений в репозитории; архитектура системы реального времени; случаи применения больших языковых моделей.

Для цитирования: Старолетов С.М. Уроки эволюции С-кода и шаблоны архитектуры реального времени на основе проекта ioquake3. Труды ИСП РАН, том 38, вып. 2, 2026 г., стр. 53-68 (на английском языке). DOI: 10.15514/ISPRAS–2026–38(2)–4.

1. Introduction

The evolution of modern compilers increasingly prioritizes rigorous code quality assurance, flagging constructs that may lead to undefined behavior or security vulnerabilities. This shift forces developers to revisit legacy codebases, often uncovering practices that once "just worked" but now violate modern security standards. Games from the late 1990s, such as Quake III Arena (1999), offer a unique perspective on this evolution. As real-time systems optimized for limited hardware, these codebases embody resource-constrained programming paradigms that remain relevant today, especially on embedded and mobile platforms.

In this article, we analyze the game engine architecture and the differences between original Quake III C code and modern refactored versions of it, examining how twenty-year-old implementations were refactored to be compilable with modern compilers such as GCC with full warnings enabled. We classify compiler-generated warnings, identify recurring anti-patterns, and describe refactoring strategies for bringing legacy code into line with modern reliability requirements.

We set ourselves the following research questions (RQ):

- **RQ1** – By examining the source code of the ioquake project, what conclusions can be drawn about its real-time architecture?

- **RQ2** – What categories of vulnerabilities and unsafe coding practices, typical of the Quake III development era (1999), have been identified by modern tools and refactored today in the ioquake3 project?
- **RQ3** – How effective is refactoring legacy code (using ioquake3 as an example) in terms of compliance with modern code quality standards?

The structure of this article is as follows. In Section 2, we provide an overview of Quake III from its history and modern view on its architecture, as well as study lessons from its real-time architecture. In Section 3, we review work related to our methods. In Section 4, we analyze the difference between the original game code written in 1999 and its current state, compilable for modern platforms. In Section 5, we check what compiler warnings the modern code can cause. We summarize in Section 6.

2. Background

2.1 An insight into Quake III

Quake III Arena (1999) is a multiplayer first-person shooter developed by id Software. It is the third game in the Quake series, but unlike previous installments, it focused entirely on online battles, abandoning a story-driven campaign. The game's history can be traced to the spirit of game studios of that era, particularly id Software, its developer [1]. It can be said that the game marked a turning point from the "Doomers" ("Doomers" are fans of the original Doom and Doom II, which became cultural phenomena of the 1990s. This generation grew up with pixelated demons, secret rooms, and modem-based multiplayer) toward “e-sports”, i.e., multiplayer gameplay against other players and AI bots [2].

The principal developer and engine architect of Quake III Arena was John Carmack. In 2005, id Software released the source code under the GPL-2.0 license, making it accessible on GitHub [3]. Subsequently, the ioquake3 project was initiated to rewrite portions of the original x86-related code, enabling compilation with modern compilers across various platforms [4].

The engine has been licensed for use in several notable games, including Call of Duty (2003) and Medal of Honor: Pacific Assault (2004). It utilized OpenGL as its primary API [5]. Carmack made a pivotal decision to abandon software rendering, opting instead to leverage graphics cards optimized for Voodoo 3 and GeForce 256, the first GPUs to support transform and lighting (T&L). The game also featured its own shaders [6] prior to the term's widespread adoption in modern graphics processing. Today, ioquake3 employs the cross-platform libSDL2, and software rendering remains available via llvmpipe for various framebuffer devices.

Some articles have analyzed the game's architecture [7], including a significant work by Fabien Sanglard, the author of a comprehensive analysis of Doom [8]. This analysis includes insights from the developers of the Quake III engine regarding their architectural decisions [9]. The engine is particularly renowned in programming circles for the "magic" Fast Inverse Square Root hack, which utilizes the constant `0x5f3759df` [10]. This optimization was crucial in the 1990s when floating-point operations were relatively slow, as it replaced numerous CPU cycles with a few efficient bitwise operations. Additionally, the engine famously implemented a 3D version of Binary Space Partitioning (BSP) [11], an algorithm that recursively divides space with planes to efficiently render static levels, detect player-wall collisions, and store lighting data, despite the labor-intensive preprocessing involved.

Let us examine the project code. At the time of development, the C99 standard had not yet been fully implemented, and the developers utilized compilers that only partially adhered to the standards of the era, often interpreting certain code elements in their own manner. The project's source code includes a project file for MS Visual C. According to a Microsoft study [12], 78% of C/C++

developers were using this compiler during that period. This compiler supported the older ANSI C standard and lacked features such as warnings for buffer overflows, analysis of uninitialized variables, monitoring of string formats, and control over type conversions. Additionally, the GCC 2.95 compiler was also in use at that time; while it was more stringent in its warnings, particularly regarding uninitialized variables, it did not offer flags like *-Wall*, *-Wextra*, or *-Wpedantic* in their current forms.

The project code had a mixture of coding styles, as exemplified by:

```
ent = &g_entities[0];
for (i=0 ; i < level.maxclients ; i++, ent++ ) { }
```

As previously mentioned, the game can now be compiled and run on Windows, Linux or Mac OS using a modern compiler, thanks to the ioquake3 port. The ability to launch such ports on modern hardware is a testament to the effectiveness of the C language in promoting project longevity and portability. It also reflects the relatively successful structure of the original code, crafted by exceptionally talented developers.

Furthermore, due to the availability of the source code and the well-developed internal world and gameplay mechanics, Quake III Arena can be regarded as a benchmark for evaluating new processor architectures, such as Elbrus and LoongArch [13] (Fig.1).

Fig. 1. The ioquake3 project running on a BT1 MIPS processor, framebuffer device (left), LoongSon’s GSGPU (center), ARM64 Apple M1 (right).

2.2 Quake III real-time architecture

According to our assumption, successful game engines can teach the correct architecture for building real-time systems to work on limited resources. To check this, we studied the current project source code [4] and found that ultimately all the game logic is calculated when processing the frame in *G_RunFrame()* from *g_main.c*. We also studied the functions called from there and thus came to an understanding of the architecture, shown schematically in figures 2 and 3.

The presented statecharts visualize the work of the main frame execution loop (*G_RunFrame())* in the game engine. Its main task is to ensure that the entire game world is processed in a strictly allotted time (about 16 ms for 60 FPS). The loop begins with a check of the global state of the level. If the level is not restarted, the system updates the global time and variables. Then the key phase occurs: iterative processing of all active entities (objects) of the game. For each entity, the following is done:

1. Cleanup of stale events to manage the lifetime of timed events (e.g., hit effects).
2. Depending on the entity type (client player, missile, item, moving platform), a highly specialized function (*G_RunMissile()*, *G_RunItem(),* etc.) is called.
3. If there is no entity-specific logic, or after it has executed, the *G_RunThink()* function is called. This is the heart of the deferred execution system.

After all entities have been processed, there is a post-processing phase where final calculations are performed for players, global game rules are checked (match end, vote timeouts). All this logic must fit into the target frame time.

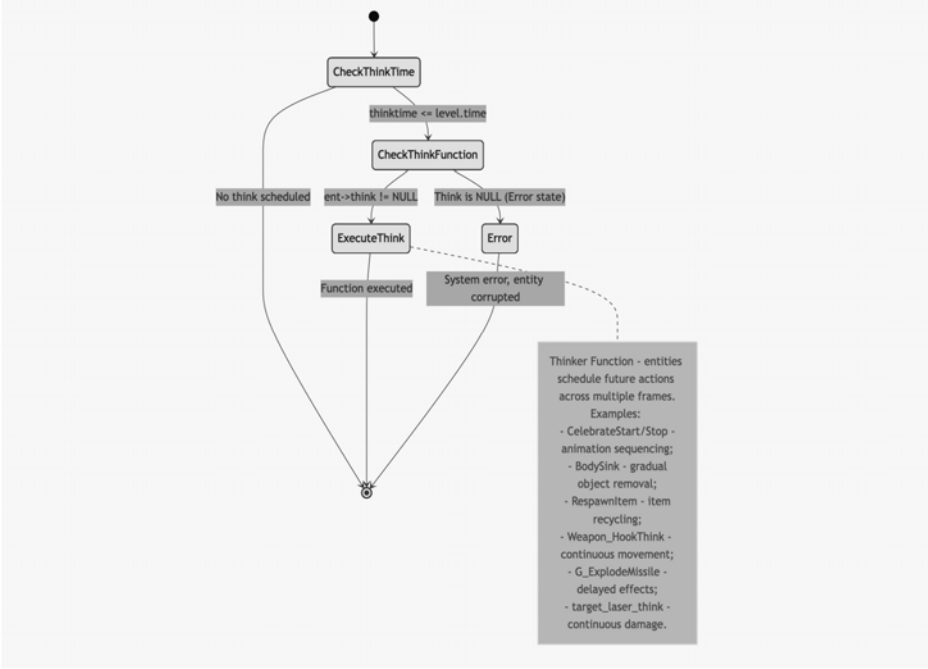

Fig. 2. Main loop (G_RunFrame()) statechart.

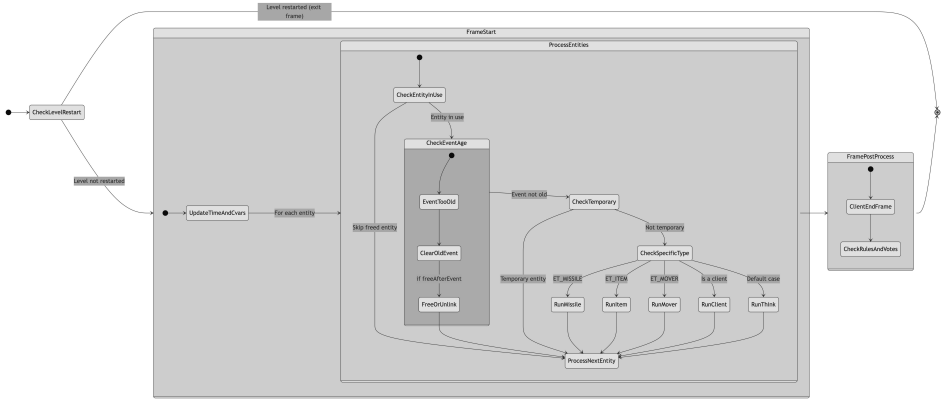

Fig. 3. Think Function (G_RunThink()) statechart – a universal pattern for all deferred actions.

The so-called *Think Functions* are an elegant solution to time management in Quake III, using approaches from real-time systems. Instead of blocking execution waiting for some event, an entity schedules a function for execution in the future. That is, one can think (execute some logic) in the allotted time intervals, but make sure that complex actions are divided into smaller ones that are planned in advance.

To schedule the execution of the code, it is enough to set

```
ent->nextthink=level.time+delay
```

and assign

```
ent->think=AFunction
```

to the game entity *ent*. During execution, the main loop *G_RunFrame()* checks every frame for each entity whether it is time to call its Think Function. This pattern allows implementing a wide variety of behaviors. Examples found in the code:

- animations: *CelebrateStart()->CelebrateStop()*;
- deleting objects: *G_Free-Entity()* with a delay nextthink=level.time+30000 (for deletion after 30 seconds);
- respawning items: *RespawnItem()*;
- continuous processes: *Weapon_HookThink*() (updates position every frame), *target_laser_think()* (calculates laser beam and deals damage every frame);
- chains of actions: *FinishSpawningItem()* (delayed completion of item spawning until the next frame, so that it is guaranteed to be in the world after its creation).

The example architecture serves as a good teaching aid for several reasons. First, it demonstrates that all logic must be executed within one frame. This teaches developers to break down tasks into small, atomic operations and always estimate their complexity to meet time constraints. Next, the presented diagrams clearly show how a large task (updating a frame) is broken down into sequential, manageable stages. This is a direct example of the architectural pattern "Batch Processing". Importantly, the considered mechanism of Think Functions is a practical implementation of cooperative multitasking within a single thread. Developers can learn from this not to use blocking calls, but to schedule operations, which is important for UI responsiveness, network interactions and any other real-time systems. Finally, the frame processing code reviewed demonstrates active memory management (see *inuse*, *neverFree*, *freeAfterEvent* flags in the code) in an environment where allocating and freeing memory at random times can lead to fragmentation and unpredictable delays. As a result, using the global level time as the basis for all calculations ensures deterministic behavior of the system.

3. Related Work

Methods for analyzing differences between software releases are frequently applied to the Linux kernel, where development relies on diffs and patches, as noted by Jiang et al. (2013) [14]. Notably, the distributed version control tool Git was specifically created for this purpose. The Spinfer tool, developed by Serrano et al. (2020) [15], utilizes semantic patch inference, a technique introduced by Andersen et al. (2010) [16]. This method identifies common clusters of identical changes and formats them in a specialized language for application across numerous drivers. However, these changes are highly specific to the Linux system code, which poses challenges for training individuals in this area.

The analysis of compiler warning messages is addressed by Sun et al. (2016) [17]. This topic is particularly relevant for the extensive Linux codebase; for instance, Melo et al. (2016) [18] compiled 42,060 kernels with all warnings enabled, analyzing 400,000 warnings to identify the most vulnerable subsystems.

In our previous work, we classified bugs based on their fixes. Specifically, we used commit messages in the Linux Git repository applying classic methods of representing text as variables and classifying them, which allowed us to identify key types of bugs in various kernel subsystems [19]. The same method can be applied to commits in cyber-physical systems repositories, but the analysis is highly dependent on the accuracy of commit messages from developers [20].

The GumTree approach by Falleri et al (2014) [21] provides fine-grained source code differencing at the AST level, including move operations, which enables accurate detection of structural changes. This approach can be effectively utilized to extract semantic change clusters from git diff output by analyzing the underlying AST modifications rather than just line-based differences. It does not directly classify or cluster changes from a raw .diff file. Its output theoretically can serve as a structured basis for further analysis to categorize change types or patterns. However, for the purposes of simply assessing changes, this approach is cumbersome.

Several books, such as the one by Feathers (2004) [22], focus on working with old or legacy code, serving as valuable resources for developers transitioning from junior to senior roles. Modern trends increasingly emphasize static analysis methods to enhance understanding of such code, as highlighted by Kirchmayr et al. (2016) [23].

In the previous section, we referenced articles that describe the architecture of Quake III. Additionally, Morisaki et al. (2019) [24] discuss automatic methods of code analysis, including an examination of the control graph with specific examples from this game.

In his 2011 article, John Carmack [25] shares his experiences with static analysis, detailing how it transformed his perspective on code quality and verification methods, which were not available during the development of the game being analyzed.

4. Analyzing Diff Between IoQuake3 Code and Original Quake III Code

4.1 Analysis methodology

In this section, we analyze the differences between the original Quake III code [3], which was open-sourced in 2005 and subsequently made available on GitHub, and its modern fork, ioquake3 [4], which can be compiled with contemporary compilers across various platforms. Since this modern version is based on the original source code, we can simply copy the new version of the source codes over the old ones and execute 'git diff' (the commit id `3fb9006e6461b877b67781cd0f0d94032804a475` of the ioquake3 project was used at the time of the experiments). Upon generating the differences in diff format (comprising 54,930 lines), we examined the changes and identified clusters of common patterns within the project code. The diff was made available on GitHub [26];

This analysis can be conducted manually or through automated methods that comprehend the semantics of the diff format. In our case, we employed an LLM (large language model) for this purpose, as these models are designed to generalize results. Specifically, we utilized DeepSeek R1 [27], and the outcomes were satisfactory by reviewing the identified clusters that were already identified by the author after a preliminary review of the code changes; this approach yielded adequate results, and relevant code examples were preserved. The prompt was simple: *"Classify the given diff file with C code changes over the years (a game project code from 1999) and suggest the main classes of changes with examples of changes made from real code, give these examples in diff format"*. The only limitation is that LLM cannot capture the entire diff, it has to be broken down into parts of approximately 4000 lines and then a final general classification is performed based on the classifications found. Notably, we could not find any published works that explicitly classified changes through diff analysis, aside from perhaps incidental uses as seen in [28, 29]. Interestingly, current applications of LLMs seem to focus more on generating patches [30, 31], which represents a different direction from our analysis. An example of our LLM query for a single diff chunk is presented in [32], and an example of a query for data summarization is presented in [33]. Everything was done through a manual series of queries, but in general, the author has the idea that it is possible to automatically split the diff of large changes, send it for analysis via, for example, the DeepSeek API, receive the results, and generate a final query based on them. The process is depicted in Fig. 4. This will give us the ability to very quickly evaluate key changes as clusters in large projects.

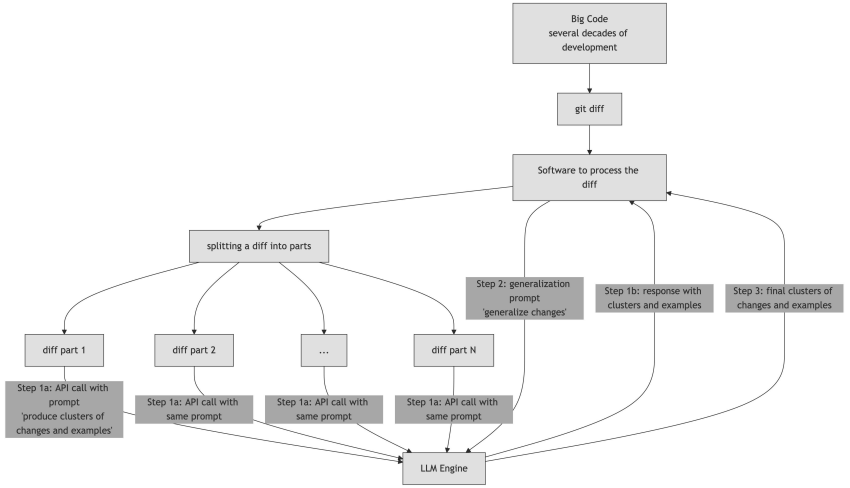

Fig. 4. LLM-driven diff analysis.

Below, we present these clusters for our project along with examples of changes for each.

4.2 Results of the general analysis of changes

String safety and overflow prevention:

```
-strncpy(cs->name, name, sizeof(cs->name));
+strncpy(cs->name, name, sizeof(cs->name)-1);
+cs->name[sizeof(cs->name)-1] = '\0';
-vsprintf(text, str, ap);
+Q_vsnprintf(text, sizeof(text), str, ap);
```

Switch to safe string handling methods to prevent vulnerabilities. Guaranteed null termination and buffer bounds checking.
Memory optimization:

```
-size += sizeof(bot_randomstring_t) + strlen(chatmessagestring) + 1;
+len = PAD(strlen(chatmessagestring) + 1, sizeof(long));
+size += sizeof(bot_randomstring_t) + len;
-double floatvalue;
+float floatvalue;
```

Reduce memory consumption through structure alignment and type changing. Critical for embedded systems and consoles.
Fixing formatting inconsistencies:

```
-Log_Write("%s", ch->filename);
+Log_Write("%s\n", ch->filename);
-SourceError(source, "out of value space\n");
+SourceError(source, "out of value space");
```

Standardize log/error output. Remove duplicate line breaks for consistency.

Refactoring conditions and naming:

```
-if (bot_developer)
+if (botDeveloper)
```

```
-if (aasword.indexessetup)
+if (!aasworld.loaded) return NULL;
```

Simplifying complex conditions and moving to semantic naming to improve readability.
Accuracy of physical calculations:

```
-if (abs(dir[0]) < 40)
+if (fabsf(dir[0]) < 40)
-phys_jumpvel = aassettings.sv_jumpvel;
+phys_jumpvel = botlibglobals.sv_jumpvel->value;
```

Replacing integer operations with floats for precise physics. Centralizing parameters through variables.
Removing dead code:

```
-void AAS_AirControl(...) {
- // Unused code
-}
```

Simplifying the code base by removing irrelevant functions. Reducing code entropy.
Cross-platform compatibility:

```
-#define MAX_PATH MAX_OPATH
+char filename[MAX_OPATH];
+#ifdef __APPLE__
+ fs_apppath = Cvar_Get(...);
+#endif
```

Adaptation of file paths and system calls for different OS (Windows/Linux/macOS).
Editing license texts:

```
-along with Foobar; if not...
+along with Quake III Arena source code...
```

Legal correctness when switching to open-source licensing.
Improving Bots AI:

```
-if (hordist < 1.6 * facecenter[2] - areastart[2])
+//if (hordist < 1.6 * (facecenter[2] - areastart[2]))
+else if (bs->inventory[ INVENTORY_PROXLAUNCHER] > 0...)
+return WEAPONINDEX_PROXLAUNCHER;
```

Fine-tuning navigation and combat behavior of bots. Adding support for new weapons.
Processing of boundary conditions:

```
-ent = &aasworld.entities[i];
+if (i < 0 || i >= aasworld.numentities) return NULL;
+ent = &aasworld.entities[i];
+if(len > sizeof(newSnap.areamask))
+Com_Error(ERR_DROP,...);
```

Protection against segmentation faults by checking array indices.
Adding new features:

```
+#ifdef USE_VOIP
+void CL_UpdateVoipIgnore(...) {
+ // VoIP implementation
+}
+#endif
```

Expanding functionality to meet modern requirements (voice chat, new input systems).
Network code optimization:

```
-void CL_AddReliableCommand(const char *cmd) {
+void CL_AddReliableCommand(... , qboolean isDisconnectCmd) {
+if ((isDisconnectCmd && unacknowledged > MAX_RELIABLE)...)
```


Improving network command handling for multiplayer stability. Overflow protection.
Modernization of the input system:

```
+("F13", K_F13),
+("PADD_LEFTSTICK_LEFT", K_PADD_LEFTSTICK_LEFT),
```

Support for modern input devices (gamepads, additional keys).
Improving file operations:

```
-void FS_Rename(const char *from, const char *to) {
+void FS_Rename(...., qboolean safe) {
+if (safe) FS_CheckFilenameIsMutable(...);
```

Safe work with files, protection from directory traversal attacks.
Architecture refactoring:

```
-for (i = 0; i < MAX_GENTITIES; i++) {
+for (i = 0; i < level.num_entities; i++) {
-typedef struct { float value; int color; } graphsamp_t;
+static float values[1024];
```

Optimization of data structures and loops. Simplification of code to improve performance.
Refactoring of header file paths:

```
-#include "../game/q_shared.h"
+#include "../qcommon/q_shared.h"
-#include "../game/be_aas.h"
+#include "be_aas.h"
```

Correction of text errors:

```
-//id Sofware BSP data
+//id Software BSP data
-Log_write("portal area %d is separating more  than two clusters\r\n", areanum);
+Log_write("portal area %d is separating more than two clusters\r\n", areanum);
```

Fixing memory leaks:

```
+FreeMemory(done);
+FreeMemory(define->name); FreeMemory(define);
```

Zeroing the variables:

```
-vec3_t beststart, bestend;
+vec3_t beststart = {0}, bestend = {0};
-float wksp[4097];
+float wksp[4097] = {0};
```

Fixing work with arrays:

```
-numareas = AAS_TraceAreas(...; sizeof(areas)/sizeof(int));
+numareas = AAS_TraceAreas(...; ARRAY_LEN(areas));
-aasworld.bboxes[i].mins[j] = LittleLong(aasworld.bboxes[i].mins[j]);
+aasworld.bboxes[i].mins[j] = LittleFloat(aasworld.bboxes[i].mins[j]);
```

4.3 Correction of mathematical operations and conversion between datatypes

Let us look separately at cases where errors in mathematics are being corrected.
Replacing 'abs' with 'fabsf' for floating point numbers:

```
-if (abs(dir[0]) < 40)
+if (fabsf(dir[0]) < 40)
-if (abs(dir[1]) < 40)
+if (fabsf(dir[1]) < 40)
```

Correct calculation of absolute value:

```
-if (abs(DotProduct(plane->normal, up)) < 0.1)
+if (fabsf(DotProduct(plane->normal, up)) < 0.1)
-if (abs(ms->origin[2] - reach->end[2]) < sv_maxbarrier->value)
+if (fabsf(ms->origin[2] - reach->end[2]) < sv_maxbarrier->value)
```

Correct calculation of projection:

```
-scale = (inventory[fs->index] - fs->value) / (fs->next->value - fs->value);
+scale = (float)(inventory[fs->index] - fs->value) / (fs->next->value - fs->value);
```

Incorrect BSP data conversion (little-endian):

```
-assworld.bboxes[i].mins[j] = LittleLong(assworld.bboxes[i].mins[j]);
+assworld.bboxes[i].mins[j] = LittleFloat(assworld.bboxes[i].mins[j]);
```

We observe that many inaccuracies found with real numbers show that due to random conversions to integers, values after the decimal point were discarded. This is apparently the reason for the jerky movements of characters, as well as incorrect interactions of the character with the environment, and teleportation effects.

4.4 Intermediate findings from the diff analysis

Overall, the changes in the code reflect a commitment to achieving stable and portable software. Given that these modifications are being made now, it is plausible that some were prompted by the use of static analyzers. We can enhance stability and reliability by addressing both subtle and obvious bugs. Our analysis focuses on the game code, where mathematical corrections are particularly relevant, and we have identified specific instances of this. As is common in C programs, the changes also addressed buffer overflow checks and NULL checks. While explicit warnings related to uninitialized values were corrected, not all were addressed, as we will discuss later. Additionally, modifications aimed at improving portability and performance optimization are evident. The code has also been cleaned up, with function changes and corrections to spelling errors in string constants, indicating that the code files were manually reviewed by developers.

Let us ask ourselves: why did older compilers (from 1999) overlook these issues? At that time, static analysis was still in its infancy. Early versions of C compilers lacked options such as *-Wall*, *-Wextra*, and *-Wpedantic*. The C language exhibited a liberal approach to types, where implicit conversions between int and float were deemed acceptable, in contrast to modern languages like Swift and Kotlin. Furthermore, the understanding of vulnerabilities was just beginning to develop, and the standard C library did not include specific protections against vulnerabilities. Concepts such as format security and buffer overflow were not prioritized, leading developers to use unsafe functions like *strcpy()* and *gets()* without verifying the length of C-strings. This issue is highlighted in Sir Tony Hoare's famous talk on billion-dollar mistakes [34].

5. Clustering warnings in the modern code

Now, let us examine the ioquake3 project code [4] (the state at the start of 2025) for the presence or absence of warnings following the recent changes. To accomplish this, we enabled the maximum warning mode (*-Wall*) in the project's Makefile. We then analyzed the compiler output, collecting all error messages. Afterward, we removed any references to specific files and code constructs from these lines. We created a hash table for each line containing errors and count the occurrences of each type of warning. This process allows us to determine how many warnings of each type remain in the source code. The results of our analysis are presented in Fig. 5.

Despite the changes observed in Section 4.2, there are still over 1,000 warnings in the current code. The majority of these warnings are of the type *-Wunused-parameter*. While this warning may seem minor, it often indicates potential risks. An unused parameter can suggest that important logic, such

as bounds checking or special case handling, has been overlooked. For instance, developers at Apple once neglected to check a signature passed as a parameter, resulting in a vulnerability [35]. In the ioquake3 code, this warning appears to reflect the ongoing evolution of the code as it is cleaned up and generalized.

Next, we encounter warnings related to comparing integer operators of different signs *[-Wsign-compare]*, which occur when an int is compared to an unsigned value. The roots of such issues can be traced back to university education, where the int data type is often used for all integers. However, this practice can lead to numerous potential errors and vulnerabilities in the code, especially in calculations, as seen in our project. It is also common for the code to return -1 to indicate an error, and if signed comparisons are incorrect, error handling may be compromised.

Another significant category of warnings pertains to missing field initializers *[-Wmissing-field-initializers]*. This occurs when a structure is initialized in the code, but not all fields are assigned values. Such situations may arise after refactoring, when a new field is added to a structure but its initialization is overlooked. This issue is a long-standing characteristic of C, in contrast to languages like Java, where object fields are automatically initialized, and it has historically led to numerous errors and vulnerabilities. However, this practice can sometimes be justified for performance reasons, particularly when the game processes vectors with some coordinates being calculated.

We also identified a small number of warnings related to unsafe pointer operations *[-Wnull-pointer-arithmetic],* which are specific to C, as well as issues with evaluating absolute values using floating-point operations for integers *[-Wabsolute-value]*, which were mentioned in the previous section but are now presented in a different context.

Overall, it is evident that while the project has undergone refactoring, it remains far from ideal.

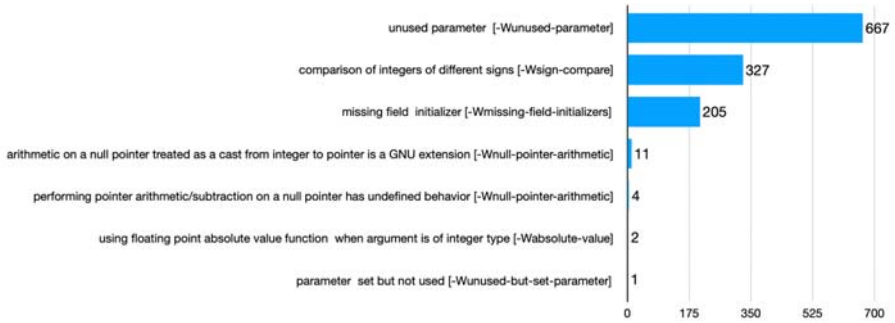

Fig. 5. GCC warnings classification in the ioquake3 project.

6. Conclusion

Computer shooters provide a means to release negative energy, and in today's world, it is preferable for such releases to occur in virtual arenas. The successful Quake III game project offers valuable lessons for modern developers through its source code. In contrast to the extensive open-source Linux kernel codebase, which is highly specific and focused on low-level programming, the game code serves as an engaging and visual teaching aid for the average developer. This article extracts essential insights from the organization and evolution of the Quake III code, emphasizing the need to enhance the quality of legacy code to meet contemporary standards.

The Quake III code was developed during a time when security and static analysis were often considered secondary concerns. Given the hardware resources available at that time, this approach was somewhat justified. However, modern compilers and tools could have mitigated many of the vulnerabilities that were identified later. The diff edits we analyzed partially address these issues, adapting the code to contemporary standards.

Overall, we did not encounter any critical errors in the original code; the changes made to the old code were primarily incremental and evolutionary, rather than revolutionary. This can be attributed to the high level of expertise possessed by the developers of that era, who were true architects of game engines, as well as the use of the classic C language, which was designed for portability and longevity. It is, however, puzzling that the same developers who implemented square root hacks using magic constants like `0x5f3759df` also made relatively basic mistakes, such as losing the fractional part of calculations (although it was possible to optimize for hardware at the expense of correctness). Unfortunately, in 1999, version control systems were not as developed as they are today, making it impossible to trace the authorship of each line of code.

The answer to **RQ1** was determined by manually reviewing the code and navigating through the functions used to render frames. We found that the developers implemented their cooperative multitasking and time-bound processing, including memory management, as is typical for game engines, in the frame rendering function. The main ideas were presented in figures 2 and 3.

The response to **RQ2** in the paper is derived from clustering the changes in the diff between the original Quake III code and ioquake3, as detailed in Sections 4.2 and 4.3. The vulnerability fixes found include correction of unsafe string handling (use of *vsprintf* -> replacement with *Q_vsnprintf*, errors in mathematical operations (*abs()* for float -> replacement with *fabs()*), array bounds checking (adding *if (i < 0 || i >= numentities)* conditions), handling of uninitialized variables (initialization of vectors with 0).

As for **RQ3**, it is addressed in the article through the analysis and classification of GCC warnings generated with the *-Wall* flag in the current version of ioquake3 (see Section 5, Fig. 5). It can be concluded that the refactoring is not fully effective: more than 1000 warnings remain. The main unresolved issues are *-Wunused-parameter* (risk of unaccounted logic), *-Wsign-compare* (signed/unsigned comparison), and *-Wmissing-field-initializers* (uninitialized structure fields). Based on our analysis, this can lead to code crashing at random times on unpopular platforms such as MIPS.

From this experience, we recognize the importance of clustering changes in the code, which can be accomplished both manually and through experiments with LLM models. Using AI to evaluate classes of changes in a large diff is justified because we do not need great precision here, but we do need generalization and examples of changes made.

Ultimately, gaining insight into the inner workings of such landmark projects allows us to explore effective solutions that can greatly enhance our understanding of programming and the development of real-time systems.

References

[1]. Kushner D. Masters of Doom: How two guys created an empire and transformed pop culture. Random House Trade Paperbacks, 2004.

[2]. Reseigh-Lincoln D. How Doom changed PC gaming forever, 2021. Available at: https://www.techradar.com/news/how-doom-changed-pc-gaming-forever, accessed 25.11.2025.

[3]. id Software. Quake-III-Arena, 2011. Available at: https://github.com/id-Software/Quake-III-Arena/, accessed 25.11.2025.

[4]. ioquake community. ioquake3, 2025. Available at: https://github.com/ioquake/ioq3, accessed 25.11.2025.

[5]. Hook B. 1999 GCC: The Quake 3 Rendering Engine, 1999. Available at: https://archive.org/details/1999-gdc-brianhookquake3, accessed 25.11.2025.

[6]. Hook B., Jaquays P. Quake III Arena. Shader Manual, 1999. Available at: https://fabiensanglard.net/fd_proxy/quake3/Q3%20Shaders.pdf, accessed 25.11.2025.

[7]. Munro J., Boldyrell C, Caplinppi A. Architectural studies of games engines-The quake series. In Proc of 2009 International IEEE Consumer Electronics Society's Games Innovations Conference. IEEE, 2009. pp. 246-255.

[8]. Sanghard F. Game engine Black Book: Doom. Software Wizards, 2018.

[9]. Sanghard F. Quake 3 source code review: architecture, 2021. Available at: https://fabiensanglard.net/quake3/, accessed 25.11.2025.

[10]. McEnirly C. The mathematics behind the fast inverse square root function code. Tech. rep, 2007. Available at: https://0x5f37642f.com/documents/McEnirlyMathematicaBehind.pdf, accessed 25.11.2025.

[11]. Tony Cricenti Hecq, Dominique and Philip Branch. 2011. Quake III Arena game structures. Available: https://figshare.swinburne.edu.au/articles/report/Quake_III_Arena_game_structures/26254658/1/files/47591027.pdf, accessed 25.11.2025.

[12]. Microsoft. Microsoft Introduces Visual C++ 6.0, 1998. Available at: https://news.microsoft.com/source/1998/06/29/microsoft-introduces-visual-c-6-0/, accessed 25.11.2025.

[13]. Independent review and test of the Loongson 3A6000 laptop, loongarch64: Linux, games, x86 emulation, GPU (In Russian). Available at: https://youtu.be/DZ9uBMin6xs, accessed 25.11.2025.

[14]. Jiang Y., Adams B., German D.M. Will my patch make it? And how fast? Case study on the Linux kernel. In proc. of 2013 10th Working conference on mining software repositories (MSR). IEEE, 2013, pp. 101-110.

[15]. Serrano L., Nguyen V., Thung F., Jiang L., Lo D, Lawall J., Muller G. SPINFER: Inferring Semantic Patches for the Linux kernel. In Proc. of 2020 USENIX Annual Technical Conference (USENIX ATC), 2020, pp.235-248.

[16]. Andersen J., Lawall J. Generic patch inference. Automated software engineering 17, 2010, pp.119-148.

[17]. Sun C, Le V., Su Z. Finding and analyzing compiler warning defects. In Proc. of the 38th International Conference on Software Engineering, 2016, pp. 203-213.

[18]. Melo J., Flesborg E., Brabrand C., Wasowski A. A quantitative analysis of variability warnings in Linux. In Proc. of the 10th International Workshop on Variability Modelling of Software-Intensive Systems, 2016, pp. 3-8.

[19]. Staroletov S.M., Starovoytov N.A, Golovnev N.A. Analyzing hot bugs in the Linux kernel by clustering fixing commit messages. Trudy ISP RAN/Proc.ISP RAS, vol 35, issue 3, 2023., pp. 215-242. EDN OWVILK.

[20]. Starovoytov N.A, Staroletov S.M. Exploring the taxonomy of commits in cyber-physical systems for enhanced error fixes investigation. Trudy ISP RAN/Proc.ISP RAS, vol 36, issue 2, 2024., pp. 33-46. EDN OSKQAQ.

[21]. Falleri J.R., Morandat F., Blanc X., Martinez M., Monperru, M. Fine-grained and accurate source code differencing. In Proc. of the 29th ACM/IEEE international conference on Automated software engineering, 2014. pp. 313-324.

[22]. Feathers M. Working effectively with legacy code. Prentice Hall Professional, 2005. Available at: https://archive.org/details/working-effectively-with-legacy-code/page/n1/mode/2up, accessed 25.11.2025.

[23]. Kirchmayr W., Moser M., Nocke L., Pichler J., Tobler R. Integration of static and dynamic code analysis for understanding legacy source code. In proc. of 2016 IEEE international conference on software maintenance and evolution (ICSME). IEEE, 2016, pp. 543-552.

[24]. Morisaki S., Kasai N, Kanamori K, Yamamoto S. Detecting source code hotspot in games software using low analysis. In proc. of 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). IEEE, 2019, pp. 484-489.

[25]. Carmack J. Static Code Analysis, 2022. Available at: https://pvs-studio.com/en/blog/posts/a0087/, accessed 25.11.2025.

[26]. Staroletov S. Source files for the article, 2025. Available at: https://github.com/SergeyStaroletov/ioquake_vs_q3, accessed 25.11.2025.

[27]. Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-Rl: Incentivizing reasoning capability films via reinforcement learning. arXiv preprint arXiv:2501.12948, 2025. Available at: https://arxiv.org/abs/2501.12948, accessed 25.11.2025.

[28]. Xing Hu, Feifei Niu, Junski Chen, Xin Zhou, Junwei Zhang, Junda He, Xin Xia, and David Lo. Assessing and Advancing Benchmarks for Evaluating Large Language Models in Software Engineering Tasks. arXiv preprint arXiv:2505.08903, 2025. Available at: https://www.arxiv.org/pdf/2505.08903v2, accessed 25.11.2025.

[29]. Rongkai Liu, Heyuan Shi, Shuming Liu, Chao Hu, Sisheng Li, Yuheng Shen, Rumzhe Wang, Nasohai Shi, and Yu Jiang. 2025. PatchScope-LLM-Enhanced Fine-Grained Stable Patch Classification for Linux Kernel. In proc. of the ACM on Software Engineering, ISSTA, 2025, pp.1513-1535.

[30]. Wei Li, Xin Zhang, Zhonghai Guo, Shaoguang Mao, Wen Luo, Guangyue Peng, Yangyu Huang, Houfeng Wang, and Scarlett Li. 2025. Fea-bench: A benchmark for evaluating repository-level code generation for

feature implementation. arXiv preprint arXiv:2503.06680. Available at: https://arxiv.org/abs/2503.06680, accessed 25.11.2025.
[31]. Zhou J. Fine-Tuning Large Language Models for Practical Software Engineering: Case Studies in Automated Patch Generation, 2024.
[32]. DeepSeek. Example clusterization of part of diff of changes (in Russian), 2025. Available at: https://chat.deepseek.com/share/i0an6j0bkpidm20ibi, accessed 25.11.2025.
[33]. DeepSeek. Final clusterization of multiple parts of changes (in Russian), 2025. Available at: https://chat.deepseek.com/share/iimmmeykoyplj4by23, accessed 25.11.2025.
[34]. Hoare T. Null references: The billion dollar mistake (March 2009), abstract of QCon London Keynote, 2009. Available at: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/, accessed 25.11.2025.
[35]. NVD, CVE-2014-1266 detail, 2014. Available at: https://nvd.nist.gov/vuln/detail/CVE-2014-1266, accessed 25.11.2025.

Информация об авторах / Information about authors

Сергей Михайлович СТАРОЛЕТОВ – кандидат физико-математических наук, доцент. Сфера научных интересов: формальная верификация, проверка моделей, киберфизические системы, операционные системы.

Sergey Mikhailovich STAROLETOV – Cand. Sci. (Phys.-Math.), associate professor. Research interests: formal verification, model checking, cyber-physical systems, operating systems.