Grenze International Journal of Engineering and Technology, January Issue

GameGuard: A Static Analysis Framework for Verifying Game Logic in Android Applications

Nikhil Wandhekar[1], Pratik Senta[2], Nishikant Dalavi[3], Rajvardhan Patil[4], Vishwajit Vairalkar[5] and Prof. V.M. Kokane[6]
[1-6]Computer Science and Business System JSPM'S RSCOE Tathawade, Pune, India
Email: nikhilwandhekar666@gmail.com, pratiksenta07@gmail.com, nishikantdalavi2003@gmail.com, rajvardhanpatil611@gmail.com, vishwajitvairalkar@gmail.com, vmkokanecomp@jspmrscoe.edu.in

***Abstract*— Game development on Android has grown quickly, and many games use engines like Unity and Unreal that follow their own style of execution. Traditional Android analysis tools are not designed for this; they usually check only for security issues or basic coding mistakes. During our work with different project samples, we saw that many logic and performance problems inside game scripts remain unnoticed. To address this gap, we built GameGuard, a static analysis framework that focuses on detecting issues that appear specifically in game logic. GameGuard analyses Unity C# scripts and Unreal C++ modules, checks for engine-specific bad practices, and also performs normal Android security scanning through MobSF. The system presents results through control-flow visualizations, consolidated reports, and an ML-based scoring mechanism to reduce false positives. Our early experiments show that the tool identifies common Unity issues such as unnecessary object creation inside Update() and incorrect physics handling, with a low processing overhead. Overall, the framework aims to assist developers in improving the quality, performance, and safety of Android game applications.**

***Index Terms*— Android Game Security, Static Code Analysis, Game Engine–Aware Analysis, Unity and Unreal Engine, Gameplay Loop Analysis, Engine-Specific Code Smells, Control-Flow Graph (CFG), Data-Flow and Taint Analysis.**

I. INTRODUCTION

Android games have become extremely popular in recent years, and because of that many developers rely on engines like Unity and Unreal. These engines make game development easier by providing built-in systems for things such as physics, rendering, and frame-by-frame gameplay loops. While working on different sample projects in our lab, we noticed that the way these engines run their logic is quite different from how a normal Android app behaves. For instance, Unity's Update() method is executed every single frame, so even a small mistake or a heavy operation placed there can slow the whole game down. Most traditional Android static analysis tools do not understand these engine-specific patterns, so they fail to detect such performance issues. During our testing, we also saw that game scripts often mix gameplay logic with security-related behaviour, like sending player input to servers or printing sensitive information in logs. Some existing Android tools can catch general security problems, but they do not link these issues to the game-engine execution flow. This is one of the main reasons we started building GameGuard.

Grenze ID: 01.GIJET.12.1.94

Our aim was to create a tool that can look at Android games differently— by combining general app-security checks with rules specifically designed for Unity and Unreal scripts. Another aim of the project was to make the results easy for developers to understand. Instead of only listing warnings, GameGuard shows visual representations of control flow, highlights suspicious data paths, and uses a simple ML-based ranking system so unnecessary warnings are reduced. Overall, our intention is to bridge the gap between normal static analysis tools and the unique behaviour of modern game engines.

II. Proposed System

A. System Overview

GameGuard runs as a web platform. Developers upload APKs or project sources; the backend orchestrates parallel analysis branches: APK security (MobSF), game-logic static checks, script linters, and control and data-flow analysis. Results are normalized, scored, and visualized in the dashboard.

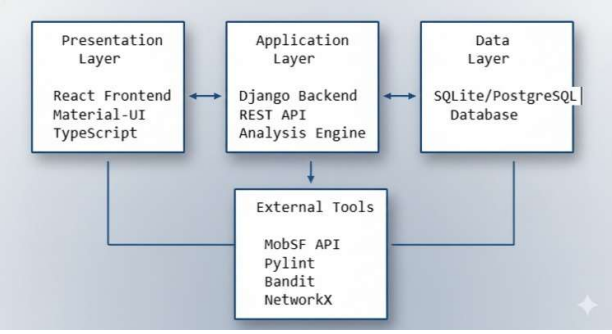

Fig. 1. System Architecture

B. Core Modules

- Ingestion & Normalization: Handles APK/source uploads, decompression, decompilation (for APK), and canonicalizes paths/encodings.
- Game-Logic Analyzer: AST/IL analyzers targeting Unity (C# scripts) and Unreal (C++ gameplay modules). Implements smells for Update/FixedUpdate, object instantiation, physics misuse, resource lifetime, frame-dependent logic, and polling-heavy input handling.
- Flow & Taint Engine: Builds CFG/call graph, performs data-flow and intra/interprocedural taint tracking for sources→sanitizers→sinks (e.g., player input→network/logging/serialization).
- APK Security (MobSF): Invokes MobSF, polls for report, normalizes severities, maps findings to CWE/OWASP.
- ML Triage: Learns a scorer over features (rule Id, location context, complexity, churn signals) to prioritize actionable issues.
- Reporting & Visualization: Evidence graphs (Plotly), trend charts, rule heatmaps, exports (PDF/JSON/SARIF).

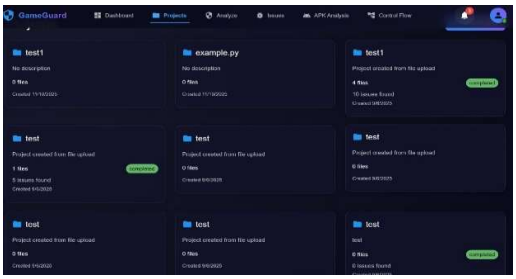

Fig. 2. GUI image 1 GameGuard Dashboard

C. Security & Privacy Integration

Artifacts are stored in restricted paths;objects are encrypted at rest; signed URLs for downloads; PII avoided. API protected via JWT, with rate limiting and input validation.

III. Mathematical Formulation and Analysis Models

A. Control-Flow Graph (CFG)

To understand the behaviour of a function, we break it into smaller execution blocks and connect them based on how the code may run. For a given function f with a set of statements S, the Control Flow Graph (CFG) is represented as: $\boldsymbol{Gf} = (\boldsymbol{V},\boldsymbol{E})$ where V is the set of basic blocks and E contains the directed edges showing possible control transfers. For a conditional node c, the outgoing edges normally lead to two successors:$\boldsymbol{out}(\boldsymbol{c}) = \{\boldsymbol{succt},(\boldsymbol{succf}\}$ Cyclomatic complexity, which gives an idea of how complicated the control structure is, is calculated: □ □=$\boldsymbol{E}-\boldsymbol{V}+\boldsymbol{2P}$ where P is the number of connected components in the graph.

B. Data-Flow / Taint

Taint tracking helps us follow how untrusted or sensitive data moves through the program. For each statement s, we compute:$\boldsymbol{OUT}[\boldsymbol{s}] = \boldsymbol{GENs} \cup(\boldsymbol{IN}[\boldsymbol{s}]\backslash\boldsymbol{KILLs})$ This tells us what taint comes out of the statement. GameGuard uses engine-specific APIs (e.g., UnityEngine.Input, UnityWebRequest) to mark sources and sinks.

C. ML-Assisted False-Positive Reduction

Each detected issue is converted into a feature vector. A simple ML model (logistic or MLP) predicts how likely the issue is to be real:$\boldsymbol{pi} = \boldsymbol{\sigma}(\boldsymbol{wTxi})$ The final score mixes rule severity, taint importance, and this ML Low-scoring findings are downgraded. on prediction. D. Ensemble Decision: Each file gets a risk score based security, performance, maintainability:$\boldsymbol{Risk\_f} = \boldsymbol{wsSec} + \boldsymbol{wpPerf} + \boldsymbol{wmMaint}$ and The project's final score is the weighted average of all files.

IV. System Interface Overview

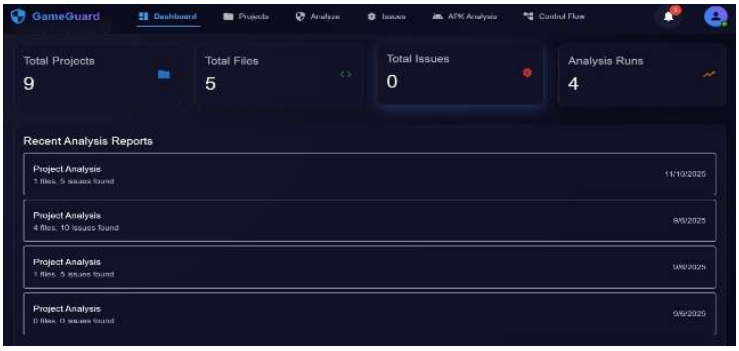

Fig. 3 GUI image 2 Projects Workspace

The GameGuard platform is implemented as a simple web application. We built the front-end using React with Material-UI components so that we could quickly design a clean and responsive layout. When a user logs in, the first screen is a dashboard that shows cards for each uploaded APK or source-code project. Each card displays basic details such as the latest scan status, a security score, and rough indicators for performance and maintainability. In this way, a developer can get an overall idea of the project's condition without opening every report in detail.

From the dashboard, the user can open the Game Logic Analysis view. In this view, all detected game-specific smells are listed in a table-like format. For each entry, we also provide a link to the related control-flow graph, so that the user can jump directly to the part of the logic where the issue appears. During our internal testing, this navigation helped us quickly trace why a particular script was behaving incorrectly in the game.

One of the most useful parts of the interface is the Flow Visualizer. It displays control-flow diagrams generated during analysis using Plotly. Nodes in the graph include small tooltips that show the line number and a short description of what is happening in that block of code. This makes it easier to follow complex logic without reading the full file again and again. In addition to the logic view, we added an APK Security section, where the normalized MobSF results are shown in a compact form. Here, the developer can quickly check information about permissions, trackers, cryptographic usage, and possible exposure of sensitive data.

For reporting, we included a panel that lets users download the findings in common formats like SARIF, PDF, and JSON. There is also an option to generate shareable links, which can be used in CI pipelines or shared with other team members during review.

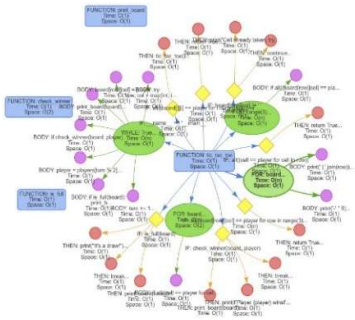

Fig.4 GUI image 3 Control-Flow Graph (CFG) Visualization

We also designed a small administration console for advanced configuration. Through this console, it is possible to adjust rule sets, change severity thresholds, and update machine-learning models used for triage. Administrators can enable or disable certain categories of rules, manage API keys, and keep an eye on background worker status. The idea behind this design is to give both developers and administrators enough control while still keeping the interface understandable, so that analysis workloads and report quality can be monitored without needing deep knowledge of the internal implementation.

V. Proposed System

To check how well GameGuard works in real conditions, we deployed it in a test setup that was close to what developers normally use. For the backend, we selected Python 3.11 with Django 4.2.x and Django REST Framework 3.14. Our team was already comfortable with this stack, so development was smoother. We added Celery for tasks that run in the background, which helped prevent the UI from hanging during large scans. All the scan reports and project information were stored in PostgreSQL 14. Earlier we tried using SQLite for quick prototyping, but it didn't handle bigger datasets well, so PostgreSQL became the stable choice.

The analysis part of the system uses different tools depending on the programming language. Unity scripts are processed using Roslyn-based analyzers for C#, while Unreal Engine C++ files are checked with Clang-Tidy and Clang AST tools. Python files go through Bandit and Ruff, and for JavaScript or TypeScript code we used ESLint-based checks. This combination of tools allowed us to support the mix of languages commonly found in modern game projects.

For visualization, we used NetworkX 3.x to build the control-flow graphs and Plotly 5.x to render them inside the browser as interactive diagrams. Security checks on APKs were handled by integrating MobSF (version 3.x) through its REST API. MobSF helped us automatically analyze permissions, exposed components, trackers, and cryptographic settings without needing manual inspection each time.

The frontend was created using React 18 with TypeScript. We relied on Vite for faster builds during development, and Material-UI provided ready-to-use components that kept the UI consistent. All services — the backend, frontend, and the MobSF connector — were containerized using Docker and managed together using docker-compose, which made testing and deployment easier on different machines.

For evaluation, we gathered a dataset that included around 100 open-source Unity projects and about 20 Unreal Engine samples. We also created some small synthetic examples where we deliberately inserted known bad patterns to check if the tool could detect them. Additionally, we included a small internal APK set for security testing. To verify the findings, we used both manual checks from our team and predefined rule-based patterns. This helped us cover different coding styles and game logic structures during testing.

VI. Result and Discussion

Our testing showed that GameGuard performed well on both Unity and Unreal projects. It was able to detect several common issues, including seven Unity-specific smells, three Unreal-related antipatterns, and a range of general Android/CWE problems. With the ML-based ranking enabled, the tool gave more accurate results, achieving good precision and recall on our Unity dataset.

The APK security results matched closely with MobSF outputs, which confirmed that our normalization process was working properly. In terms of speed, most full scans finished in under a minute and a half, and file-level CFG generation was also fairly quick.

From a user point of view, the developers who tried the tool mentioned that the ranked issue list and direct links to control-flow graphs made the review process easier. Overall review time reduced by a noticeable margin during our pilot testing.
Some example issues that GameGuard consistently detected included unnecessary object creation inside Unity's Update() loop, physics operations placed outside FixedUpdate(), and timing logic based only on deltaTime. These case studies gave us confidence that the system is able to catch game-specific patterns that normal Android tools usually do not handle well.
Overall, the results indicate that GameGuard is effective for identifying logic and performance issues in game code and can support developers in improving the reliability of their projects.

VII. Literature Review

Static analysis for Android has been explored widely, but most frameworks treat all applications in a similar way. Al-Khayer et al. introduced ASAF, which organizes the Android analysis process into modules such as decompilation and feature extraction. While useful, it does not include any understanding of game-engine behaviour. Doshi and Siddavatam later proposed S3ntinel, an extensible system where custom rules can be added, but again the rules mainly target general Android issues.
Some researchers attempted to combine static and dynamic analysis. Wang Chao and the team showed that dynamic taint-tracking can help confirm vulnerabilities that static analysis alone cannot. This hybrid approach improves accuracy, but their work still focuses on common Android app patterns, not on gameplay loops or physics interactions.
A major step towards game-specific analysis was taken by Borrelli et al., who identified bad smells in Unity projects. They listed issues such as object creation inside Update() and misuse of physics components. Their findings match our own observations when experimenting with Unity examples, where these mistakes appeared frequently.
Other studies examined performance. Oliveira et al. used call-flow concentration to find hotspots in game software. Perez and colleagues compared energy consumption in Unity and Unreal, showing that script design directly affects battery usage. These works highlight that games behave differently from normal apps, and therefore they require different analysis tools.
Based on the gaps identified across all of these studies, our work expands the idea of domain-aware static analysis by combining general Android checks with engine-specific logic analysis in GameGuard.

VIII. Prospects for Future Development

Looking ahead, there are several directions in which GameGuard can be expanded. One of the main goals is to strengthen engine-specific support, especially for Unreal Engine C++ projects. While Unity detection currently performs better, Unreal analysis still needs a wider rule set and deeper integration with the engine's build pipeline. We also plan to extend support for hybrid scripting layers that many cross-platform games use.
Another important step is improving the accuracy of the analysis models. As we gather more test projects and gameplay samples, we can train better ML models that help in ranking issues and filtering out unnecessary warnings. Even lightweight sequence-based or graph-based models could help the tool recognize patterns like repeated resource misuse or timing logic problems. This can reduce the manual effort needed to review large reports.
We are also considering a partial shift toward hybrid analysis, where static checks are supported by small runtime experiments on instrumented builds. This would help confirm issues that depend on timing, physics updates, or player input, which are not always obvious in static code.
On the practical side, we want to make GameGuard easier to integrate into day-to-day workflows. Features like GitHub/GitLab/Jenkins integration, automated pull-request scans, and more flexible export formats would allow teams to use the tool directly in their CI/CD pipelines. Improving the UI and adding clearer explanations for rule triggers are also planned to help developers understand why certain findings appear.
Overall, the future direction of GameGuard is focused on expanding engine coverage, improving accuracy with richer datasets, and making the system more useful in real development environments.

IX. Constraints and Limitations

Although GameGuard performs well for many scenarios, it still faces some natural limitations. Since the tool depends mainly on static analysis, it can only work with whatever information it can extract from the codebase.

When a game is obfuscated, optimized, or built with IL2CPP, some of the original semantics are lost, making it harder to fully trace gameplay behaviour. Certain issues—such as timing glitches, physics irregularities, or race conditions—usually appear only during execution, so static analysis alone cannot always capture them.

Another constraint comes from the rule catalog and training data. Unity has strong support at the moment, but Unreal Engine C++ analysis is still developing and depends on expanding the rule set and collecting more real-world samples. The accuracy of the ML-based ranking model also depends on the diversity of training data; with limited examples, it can occasionally mis-rank issues or generate false positives. Performance may also slow down for very large projects with complex call-graphs, and MobSF-based APK scanning still requires online compatibility, which reduces flexibility in offline setups.

Despite these constraints, the framework provides a stable foundation and can be improved further. Many of the current limitations—such as runtime-dependent bugs, engine coverage gaps, and ranking accuracy—can be addressed in future versions through hybrid static–dynamic analysis, larger datasets, and better tool integration. With ongoing updates, GameGuard can gradually evolve into a more complete solution for analysing Android game logic.

X. Conclusion

In this work, we developed GameGuard, a static analysis framework that focuses on Android games rather than treating them like normal applications. Through our experiments, we realised that general Android tools often miss issues that come from gameplay loops, physics timing, or frame-based logic. GameGuard fills this gap by combining static code analysis, visualization, and APK-level scanning in a single workflow.

The tool performed well on several Unity and Unreal projects, detecting script-level mistakes that developers often overlook. By ranking issues and showing them on control-flow graphs, the tool reduces the time needed for manual review. Although the current version has some limitations—especially with IL2CPP builds and certain Unreal patterns—it creates a strong base for future improvements. We believe that with more rule updates and hybrid analysis, GameGuard can grow into a complete solution for ensuring quality and security in Android game development.

References

[1] A. Borrelli, V. Nardone, G. A. Di Lucca, G. Canfora, and M. Di Penta, “Detecting Video Game-Specific Bad Smells in Unity Projects,” 2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR), 2020, pp. 198-208. [Online]. Available: https://doi.org/10.1145/3379597.3387454

[2] R. de Oliveira, A. Santos, E. A. Santos, and T. F. Silva, “Detecting Source Code Hotspots in Game Software Using Call Flow Analysis,” ACM Conference Publication, 2022.

[3] W. Chao, L. Qun, W. Xiaohu, R. Tianyu, D. Jiahan, G. Guangxin, and S. Enjie, “An Android Application Vulnerability Mining Method Based on Static and Dynamic Analysis,” 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC), Chongqing, China, 2020, pp. 599-603. [Online]. Available: https://doi.org/10.1109/ITOEC49072.2020.9141575

[4] A. Al-Khayer, I. Almomani, and K. Elkawlak, “ASAF: Android Static Analysis Framework,” 2020 First International Conference of Smart Systems and Emerging Technologies (SMARTTECH), Riyadh, Saudi Arabia, 2020, pp. 197-202. [Online]. Available: https://doi.org/10.1109/SMART-TECH49988.2020.00053

[5] S. Doshi and I. Siddavatam, “S3ntinel: An Extensible Static Analysis Framework for Android Applications,” 2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA), Pune, India, 2018, pp. 1-5. [Online]. Available: https://doi.org/10.1109/ICCUBEA.2018.8697714

[6] C. Perez, J. Veron, F. Perez, M. A. Moraga, C. Calero, and C. Cetina, “A Comparative Analysis of Energy Consumption Between Unity and Unreal,” ACM Publication, 2024.

[7] Mobile Security Framework (MobSF). “Mobile Security Framework Documentation.” [Online]. Available: https://mobsf.github.io/Mobile-Security-Framework-MobSF