Eclipse Evolution: Returning to First Principles

A Technical Whitepaper on Architectural Security in Game Automation

Eclipse Development Team

October 2025

Version 1.0

Abstract

This whitepaper presents a comprehensive analysis of architectural security considerations in game automation software development. We examine the evolution of the Eclipse project through three distinct phases: an initial systems-level implementation, a Python-based rapid development phase, and the current return to low-level architecture. Through detailed technical analysis, we demonstrate why Python-based solutions—even when compiled with modern toolchains—fundamentally cannot achieve the security characteristics required for stealth operation in adversarial environments.

Contents

1 Introduction

Eclipse represents a development effort in game automation technology that began in January 2025. This document chronicles the architectural decisions made throughout the project lifecycle, with particular emphasis on the security implications of language and runtime choices.

We announce a fundamental architectural shift as Eclipse returns to its original low-level foundation—a decision driven by empirical evidence regarding the incompatibility between rapid iteration frameworks and operational security requirements.

2 Architectural Evolution

2.1 Phase 1: Original Systems-Level Implementation (January–May 2025)

Eclipse began as a systems-level application built on a hybrid architecture combining C, C++, and Rust. This architecture was selected specifically for two primary characteristics:

1. **Stealth**: Minimal process fingerprinting surface
2. **Performance**: Native code execution with zero runtime overhead

The original implementation provided exceptional security characteristics, but development velocity presented significant challenges. Every feature addition required careful memory management, manual optimization, and extensive testing cycles.

2.2 Phase 2: Python Rewrite for Rapid Development (May–August 2025)

In May 2025, facing pressure to rapidly iterate on detection algorithms, movement systems, and game logic, a strategic decision was made to rewrite the core engine in Python. The benefits were immediate:

- Feature development velocity increased 5–10×
- Computer vision prototyping with OpenCV/NumPy became trivial
- Algorithm experimentation required minutes instead of hours
- Hot-reloading enabled real-time testing without recompilation

For research and development purposes, Python proved transformative. However, as the userbase expanded and detection systems evolved, the architectural debt became untenable.

2.3 Phase 3: Return to Systems Programming (August 2025–Present)

After four months of Python development and experimentation with Nuitka compilation in August 2025, the decision was made to return to the original low-level architecture—this time with battle-tested algorithms and a mature feature set.

3 Python Security Constraints

3.1 Standard Python Runtime Vulnerabilities

The Python implementation, despite enabling rapid development, created inherent fingerprinting vulnerabilities:

- Process memory exposes `python3XX.dll` interpreter

- Dependency tree trivially extracted via process inspection (numpy, opencv, win32api)
- Module paths stored as plaintext strings: `src.detection.detection`, `src.routine.routine`
- PyObject heap patterns create unique memory signatures
- Import tables reveal entire technology stack
- GIL serializes operations despite threading, adding detectable mutex patterns

3.2 Detection Surface Analysis

Process enumeration via Windows Task Manager immediately reveals:

```
Process -> DLLs:
|-- python313.dll        [Python interpreter detected]
|-- opencv_world480.dll  [Computer vision pipeline exposed]
|-- numpy.core.pyd       [Data processing framework exposed]
|-- win32api.dll         [Input system exposed]
+-- [20+ additional Python packages visible]
```

This represents an unacceptable security posture for adversarial environments where anti-cheat systems actively enumerate process dependencies.

4 Nuitka Compilation Analysis

4.1 Attempted Mitigation Strategy

Nuitka compilation was explored as a potential middle ground—attempting to preserve Python's development velocity while eliminating interpreter artifacts.

4.2 Successful Mitigations

Nuitka compilation successfully addressed several Python runtime issues:

- Removed `python313.dll` from process memory
- Eliminated `.pyc` bytecode structures
- Provided 10–30% performance improvement
- Prevented trivial script decompilation

4.3 Persistent Vulnerabilities

However, Nuitka compilation could not resolve fundamental architectural constraints:

4.3.1 External C Extension Dependencies

- OpenCV remains loaded as `opencv_world4XX.dll`
- NumPy arrays require `.pyd` shared libraries
- Computer vision pipeline fully exposed via DLL enumeration
- Identical fingerprinting vulnerability to standard Python, minus interpreter

4.3.2 Embedded Python Runtime Artifacts

- libpython bundled into binary (∼20MB of detectable code patterns)
- GIL (Global Interpreter Lock) implementation signatures in memory
- PyObject allocation patterns detectable via heap analysis
- Python C-API symbols (`PyDict_SetItem`, `PyTuple_New`, `_Py_Dealloc`) present unless aggressively stripped

4.3.3 Known Compiler Signatures

- Nuitka "onefile" mode extracts to `%Temp%\onefile_<PID>_<timestamp>` paths
- Zstd-compressed resource unpacking during startup (detectable via filesystem monitoring)
- Security researchers have documented extraction tools specifically targeting Nuitka binaries
- Antivirus vendors flag Nuitka compilation patterns due to widespread abuse by malware authors (ApolloRAT, TROX Stealer)

4.3.4 Performance Limitations

- FFI overhead on every OpenCV call (Python heap to native memory copies)
- GIL remains active, serializing multi-threaded operations
- Input latency remains 50–100μs per event versus sub-microsecond native code

4.4 Assessment

Nuitka provided security theater rather than security architecture. Detection systems do not require finding `python.exe`—they enumerate DLLs, scan heap patterns, monitor filesystem extraction, and analyze import tables. Nuitka obscures the interpreter but cannot eliminate Python's architectural fingerprints without complete replacement.

5 Low-Level Architecture

5.1 Technology Stack

The current implementation utilizes a hybrid systems programming stack:

- **C/C++**: Direct Win32 API integration, OpenCV native compilation, SIMD-optimized image processing
- **Rust**: Memory-safe async runtime, zero-cost abstractions, compile-time correctness guarantees
- **Native Compilation**: LLVM backend with LTO (Link-Time Optimization) and CPU-specific instruction sets

5.2 Strategic Rationale

The Python phase achieved its purpose: validation of every algorithm, refinement of every detection system, and perfection of every movement pattern. This knowledge is now being ported to a foundation that cannot be compromised by architectural limitations.

6 Technical Advantages

6.1 Complete Dependency Erasure

```
Python/Nuitka:              Low-Level Architecture:
|-- python3XX.dll (removed) |-- kernel32.dll (Windows system)
|-- opencv_world.dll        |-- ntdll.dll (Windows system)
|-- numpy.pyd               +-- [no third-party signatures]
|-- win32api.dll
+-- libpython (embedded)
```

All dependencies (OpenCV, async runtime, math libraries) are compiled directly into the binary as native machine code. Zero external DLLs. Zero shared libraries. Zero fingerprints.

6.2 Memory Layout Characteristics

Python/Nuitka:

- PyObject reference counting
- GIL mutex patterns
- Embedded runtime signatures

Low-Level Architecture:

- Generic allocator patterns
- Stack-allocated operations
- No interpreter artifacts

Memory scanning reveals nothing distinctive—the process could be Discord, Steam overlay, or any native application.

6.3 Performance Characteristics

- Zero-cost abstractions: Ownership system compiles to identical assembly as hand-written C
- Static dispatch: No vtable overhead, all polymorphism resolved at compile-time
- SIMD auto-vectorization: Compiler optimizes loops to AVX2/SSE4 instructions
- Lock-free concurrency: Work-stealing scheduler, no GIL serialization
- Hardware timing: RDTSC-based profiling for sub-microsecond input precision

6.4 Anti-Detection Architecture

Python/Nuitka:

- Known extraction patterns
- Temporary folder artifacts
- Library DLLs

Low-Level Architecture:

- Single binary
- Minimal imports
- Embedded resources in `.rodata` section
- Symbol stripping

7 Comparative Analysis

7.1 Detection Surface Comparison

Attack Vector	Python	Nuitka	Low-Level
Process DLL enumeration	Instant detection	CV libs exposed	Clean (system only)
Module path scanning	Plaintext strings	Partially obfuscated	None (compiled out)
Heap pattern analysis	PyObject signatures	libpython patterns	Generic allocator
Import table inspection	10+ libraries	Embedded runtime visible	Minimal (2 system DLLs)
Temp file extraction	None	Onefile artifacts	None
Runtime overhead	50–500μs/op	10–100μs/op	<1μs/op
AV false positives	Moderate	High (malware stigma)	Low
Code extraction	Trivial (.py files)	Difficult (C code)	Requires RE tools

Table 1: Detection surface comparison across implementations

7.2 Security Implementation Comparison

Python/Nuitka Detection (Trivial):

```
// Anti-cheat pseudocode
if (GetModuleHandle("opencv_world*.dll") ||
    GetModuleHandle("python*.dll") ||
    DetectTempExtraction("onefile_*") ||
    ScanHeapForPyObject() ||
    EnumerateSymbols("Py*")) {
    FlagAsBot();
}
```

Low-Level Detection (Requires Binary Reversing):

```
// Anti-cheat pseudocode
if (/* ...no distinctive patterns exist... */) {
    // Process appears as generic native application
    // Must reverse engineer entire binary to identify behavior
    // No shortcuts via DLL enumeration or memory signatures
}
```

The difference represents categorical elimination of vulnerability classes, not incremental improvement.

8 Virtualization and Sandboxing Considerations

8.1 Design Philosophy: Unified Application Architecture

Eclipse's design philosophy emphasizes a **unified application architecture** where all functionality resides within a single process, rather than relying on external software, virtual machines,

or hardware-based isolation.

8.1.1 Rationale Against VM-Based Approaches

While virtualization technologies (VirtualBox, VMware, QEMU) provide process isolation, they introduce significant drawbacks for game automation:

- **Performance Degradation**: Graphics acceleration and input latency suffer under virtualization
- **Detection Surface**: Hypervisor artifacts (VM exit handlers, CPUID leaves, timing discrepancies) are trivially detectable
- **Operational Complexity**: Multi-system architecture increases maintenance burden and attack surface
- **User Experience**: Additional software dependencies reduce accessibility

8.1.2 Native Sandboxing: Future Considerations

The unified architecture does not preclude future sandboxing implementations. Eclipse's roadmap includes:

- **Process-level isolation**: Potential use of Windows AppContainer or Linux namespaces for compartmentalization
- **Unix compatibility**: Following Nexon's announcement of MapleStory availability on macOS, cross-platform sandboxing strategies are under evaluation
- **Deferred implementation**: Sandboxing will be considered post-Unix port, leveraging platform-native security primitives rather than external virtualization

The core principle remains: *security through architectural design, not operational overhead.*

9 Performance Benchmarks

Internal testing comparing all three implementations:

Metric	Python	Nuitka	Low-Level	Improvement
CV Pipeline	8.0ms	6.2ms	2.3ms	3.48× vs Nuitka
Input Latency	500μs	80μs	0.6μs	133.3× vs Nuitka
Memory Usage	180MB	95MB	48MB	1.98× vs Nuitka
Startup Time	3.2s	1.1s	0.28s	3.93× vs Nuitka
DLL Count	23	8	2	4.0× vs Nuitka

Table 2: Performance benchmarks across all implementations

10 Lessons Learned

10.1 Python's Contributions

- Rapid algorithm validation and iteration

- Extensive testing of detection methods
- Community feedback on feature sets
- Battle-tested movement and CV systems

10.2 Python's Limitations

- Architectural security against modern detection systems
- Performance parity with native code
- Clean process memory footprint
- Protection from DLL enumeration

10.3 Development Methodology

1. **Phase 1 (January–May 2025)**: Low-level foundation, slow feature development
2. **Phase 2 (May–August 2025)**: Rapid Python prototyping, algorithm refinement, security compromises accepted
3. **Phase 3 (August 2025–Present)**: Return to low-level with mature algorithms—optimal outcome

10.4 Open Source Python Release

The Python codebase developed during Phase 2 will be released as open-source reference material in the future. This release will provide the community with:

- Fully functional detection algorithms and computer vision implementations
- Movement system architectures and pathfinding logic
- Educational examples of game automation techniques
- Comparative baseline for understanding architectural security tradeoffs

The open-source release serves both educational purposes and transparency regarding the technical foundations upon which the current low-level implementation was built.

11 Conclusion

Nuitka compilation represents the maximum security achievable while retaining Python's runtime. For projects where this is acceptable, it remains a reasonable choice. However, for Eclipse's threat model—where detection systems actively enumerate DLLs, scan memory patterns, and monitor filesystem behavior—Python's architectural constraints are unfixable without complete replacement.

The Python detour was chosen deliberately, extracted maximum value from rapid iteration, and the project now returns to first principles with battle-hardened algorithms.

Eclipse: built on C/C++/Rust, proven by Python, invisible by design.

12 References

[1] Nuitka Binary Extraction Tools. GitHub. https://github.com/extremecoders-re/nuitka-extractor

[2] ApolloRAT Malware Analysis (Nuitka-compiled). Cyble Research. https://cyble.com/blog/apollorat-evasive-malware-compiled-using-nuitka/

[3] TROX Stealer Analysis. December 2024.

[4] Python Runtime Fingerprinting Research. Various sources.

A Legal Disclaimer

A.1 Trademark and Affiliation Notice

Eclipse is an independent software project and is **not affiliated with, endorsed by, or connected to** any of the following entities:

- Nexon Co., Ltd.
- Nexon Korea Corporation
- Nexon America Inc.
- Nexon GT Corporation
- Nexopace (Nexpace)
- Wizet Corporation
- Neople Inc.
- Any other Nexon subsidiary or affiliated entity

Eclipse is not affiliated with, endorsed by, or connected to MapleStory, MapleStory N, MapleStory Worlds, or any MapleStory-related intellectual property.

All trademarks, service marks, trade names, trade dress, product names, and logos referenced in this document are the property of their respective owners. References to third-party software, games, or companies are made solely for technical analysis and educational purposes.

A.2 Intended Use

This whitepaper is provided for educational and research purposes only. The technical analysis presented herein is intended to advance understanding of software security architecture and is not intended to facilitate violations of terms of service, end-user license agreements, or applicable laws.

A.3 No Warranty

This document is provided "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, or non-infringement.

B Metadata

Document Version:	1.0
Last Updated:	October 2025
Classification:	Public Technical Documentation
Authors:	Eclipse Development Team
Contact:	[Technical documentation contact information]

The Python codebase developed during Phase 2 (May–August 2025) will be released as open-source reference material for educational purposes. Production deployment exclusively uses the low-level architecture.