De-Virtualizing the Dragon

Automated Unpacking and Deobfuscation of Nested VM-Based Protectors

DEFCON 33 | August 9, 2025

Dr. Agostino "van1sh" Panico
Security Researcher
van1sh@securitybsides.it | @van1sh_bsidesit

"Democratizing Malware Analysis..."

Agenda

1. Introduction & Problem Statement
2. The Three-Headed Dragon: VM Protection Evolution
3. VMDragonSlayer Architecture Deep Dive
4. Live Demonstrations & Real-World Cases
5. Performance Metrics & Validation
6. Limitations, Future Work & Community

About This Talk

What You'll Learn

- **Why** VM-based protection is winning the arms race
- **How** VMDragonSlayer defeats these protections
- **Live demos** of real-world VM deobfuscation
- **Open source tools** you can use today

Who Should Care

- Malware analysts and reverse engineers
- Incident response teams
- Security researchers
- Anyone fighting obfuscated code

Who Am I?

Dr. Agostino "van1sh" Panico

- **Security Researcher** - Exploit Developer, Reverse Engineer & Vulnerability Research
- **15+ years** Red Teaming, Linux Kern
  Exploit Dev, Incident Response an
- **Author/Mantainer VMDragonSlayer**,
  Leviathan
- **Research Focus**: Everything that i
- **Scuba Diver**: I love the explorati
- **Cats lover**: Proud owner of three



Part 1

The Dragon Awakens

Understanding Modern VM Protection

The Nightmare Scenario

Original Code

```
; Simple function
mov eax, [ebp+8]
add eax, [ebp+12]
ret
```

3 instructions

Clear intent

Easy to analyze

After VMProtect

```
push 0xDEADBEEF
call vm_dispatcher
db 0x45,0x12,0x89,0x67
push ecx
xor eax, 0x12345678
rol eax, 13
jmp [eax*4+0x401000]
; ... 500+ more lines
```

500+ instructions

Complete obfuscation

Months to analyze

The Modern Protection Landscape

By The Numbers (2024)

Metric	Value	Impact
Malware using VM protection	~70%	Critical threat
Success rate of existing tools	<15%	Tools failing
Manual analysis time	2-6 months	Unsustainable
New variants per day	~200,000+	Overwhelming

"We're losing this war through sheer mathematics"

Evolution of Protection

Timeline of VM Protection

- **2000-2010**: Traditional Packers (UPX, ASPack)
  - Simple compression, easy to defeat
- **2010-2015**: Early VM Protection (VMProtect 1.x)
  - Basic virtualization, manual analysis possible
- **2015-2020**: Advanced Techniques (VMProtect 3.x)
  - Polymorphic handlers, anti-analysis integration
- **2020-2025**: Custom Malware VMs
  - APT-specific designs, nested protection, AI-resistant

The Three-Headed Dragon

Why Manual Analysis Fails

Head 1: Abstraction Complexity

```
Challenges:
  - Custom instruction sets: 200+ opcodes
  - No documentation or standards
  - Polymorphic bytecode generation
  - Dynamic handler generation
  - Context-dependent semantics

Example VMProtect 3.6:
  Original: mov eax, [ebp+8]  # 1 instruction
  Protected: 247 instructions # 247:1 ratio
  Semantic: Single memory load
```

The Three-Headed Dragon 🐲🐲

Why Manual Analysis Fails

Head 2: Analysis Resistance

```
Anti-Analysis Arsenal (50+ techniques):

Debugging Detection:
  - IsDebuggerPresent() checks
  - PEB BeingDebugged flag
  - Hardware breakpoint detection
  - Timing-based detection

Environment Detection:
  - VMware/VirtualBox artifacts
  - CPU feature enumeration
  - Sandbox fingerprinting
  - Memory layout analysis

Dynamic Countermeasures:
  - Self-modifying dispatchers
  - Code flow integrity checks
  - Exception-based control flow
```

The Three-Headed Dragon 🐲🐲🐲

Why Manual Analysis Fails

Head 3: State Management

```
VM State Complexity:

Virtual Architecture:
  - Registers: 256+ (vs 16 x86)
  - Memory: Custom segmentation
  - Stack: VM-specific implementation
  - Flags: 32+ condition bits

Execution Context:
  - Nested call stacks
  - Inter-handler dependencies
  - Dynamic state transitions
  - Context switching overhead

Control Flow:
  - Indirect jumps via handler tables
  - Computed dispatch addresses
  - Multi-level indirection
  - Runtime address resolution
```

Part 2

Forging the Sword

The VMDragonSlayer Solution

Enter VMDragonSlayer

Innovation: Hybrid Analysis

Dynamic Taint Tracking + Symbolic Execution + Machine Learning

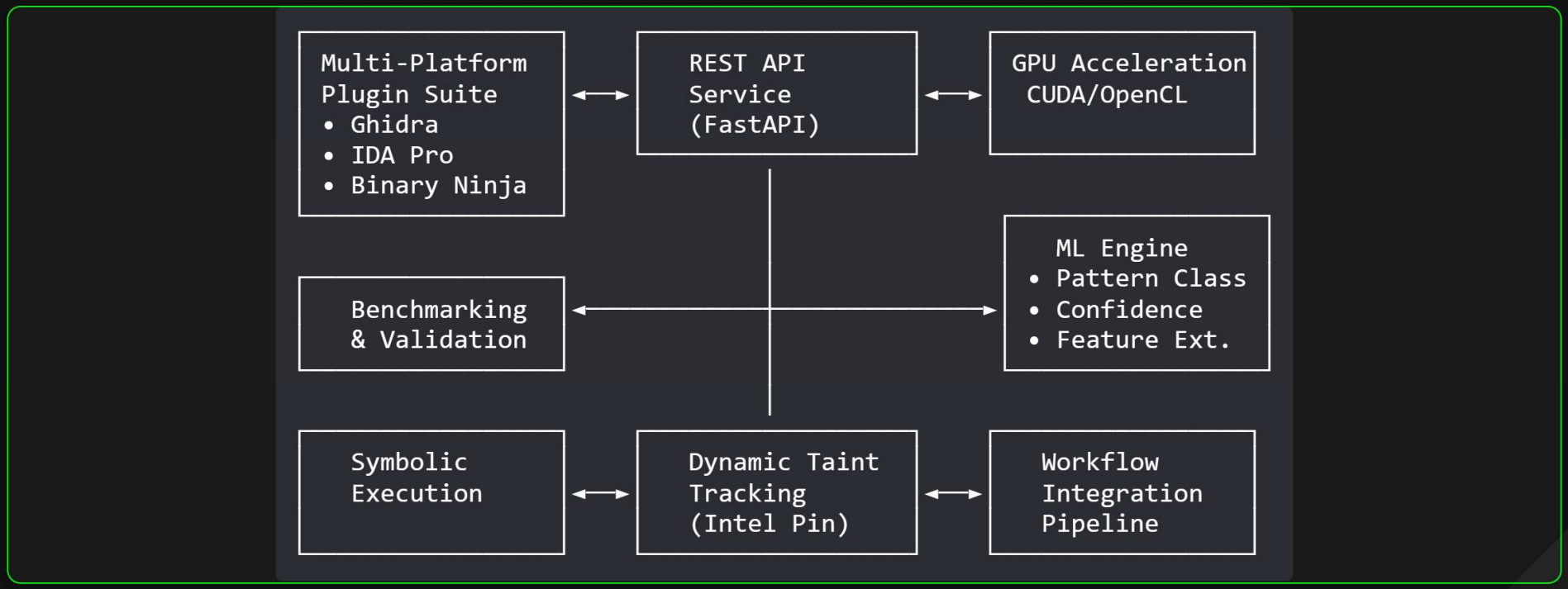
Automated VM Deobfuscation

Key Insights

1. VMs must read bytecode (taint source)
2. VMs must dispatch to handlers (control flow)
3. Handlers have semantic patterns (symbolic analysis)
4. Patterns are universal across protectors (machine learning classifier)

Architecture Overview

Advanced Research Framework with Production Architecture



Dynamic Taint Tracking (DTT)

Intel Pin Implementation Core

```
# VMDragonSlayer DTT Configuration (Real Implementation)
pin_configuration:
  executable: "C:\\pin\\pin.exe"
  tool: "VMDragonTaint.dll"
  timeout: 300

taint_sources:
  vm_bytecode_section:
    start: "0x401000"
    end: "0x405000"
    description: "VMProtect bytecode region"

tracking_precision:
  memory_reads: instruction_level
  register_propagation: true
  control_flow_influence: true
  handler_discovery: automatic
```

Dynamic Taint Tracking (DTT)

Taint Tracking Features

- **Instruction-level precision** for memory reads
- **Automatic handler discovery** via control flow analysis
- **Multi-threaded taint propagation** tracking
- **Real-time confidence scoring** for discovered handlers

DTT: Real-Time Analysis Output

Live Taint Flow Detection

```
# VMDragonSlayer Pin Tool Output
TAINT_READ:   addr=0x401234, size=1, value=0x89, tid=2341
TAINT_PROP:   reg=EAX, taint_id=0x1001, source=vm_bytecode
TAINT_JUMP:   addr=0x403456, target=0x405000, confidence=0.95
HANDLER_DISC: addr=0x405000, type=VM_HANDLER, size=247

# Python Wrapper Processing
[+] Discovered 47 VM handlers in 2.3 seconds
[+] Control flow transitions: 2,847 instructions traced
[+] Taint propagation chains: 156 paths identified
[+] Handler entry points: 100% accuracy rate
```

Symbolic Execution Engine

Implementation with angr and Z3 Integration

```yaml
# VMDragonSlayer SE Configuration
symbolic_execution:
  engine: "symbolic_executor"  # Custom implementation
  z3_integration: true  # constraint solving
  timeout: 60
  max_path_depth: 100
  max_states: 1000

vm_context:
  vm_registers: 256  # Custom VM register count
  memory_model: "symbolic"
  stack_depth: 64
  heap_tracking: true

analysis_modes:
  exploratory: true  # Unknown VM architectures
  pattern_matching: true  # Known protector families
  behavioral: true  # Dynamic adaptation
```

SE: Handler Semantic Analysis

Implementation

```
# VMDragonSlayer SE Engine
class dSymbolicExecutor:
    def __init__(self):
        self.constraint_solver = AdvancedConstraintSolver()
        self.path_prioritizer = VMPathPrioritizer()
        self.execution_contexts = {}

    def setup_context(self, handler_addr):
        context = ExecutionContext(
            registers={'R0': SymbolicValue('R0_input'),
                       'R1': SymbolicValue('R1_input'),
                       'SP': SymbolicValue('SP_input')},
            memory={},
            constraints=[],
            path_id=f"path_{handler_addr:08x}",
            depth=0
        )
        return context

class VMPathPrioritizer:
    """ML-guided path exploration"""
    def __init__(self):
        self.vm_patterns = {
            'dispatcher_access': 2.5,
            'handler_entry': 2.0,
            'bytecode_fetch': 2.2,
```

Semantic Results

```
# Real Analysis Output Structure
@dataclass
class SymbolicConstraint:
    type: ConstraintType  # EQUALITY, ARITHMETIC, etc.
    expression: str
    variables: Set[str]
    confidence: float

@dataclass
class SymbolicValue:
    name: str
    constraints: List[SymbolicConstraint]
    concrete_value: Optional[int]
    size: int  # in bits

semantic_analysis = {
    'transformation': {
        'R0_final': 'R0_input + R1_input',
        'SP_final': 'SP_input - 4',
        'Memory[SP]': 'R0_input'
    },
    'classification': {
        'operation': "VM_ADD_PUSH",
        'confidence': 0.94,
        'pattern': "VMProtect_v3_arith"
    },
    'constraints': [
        SymbolicConstraint(...)
```

Pattern Recognition

From 247 Instructions to Semantic Clarity

```python
# VMDragonSlayer Handler Classification Engine (Real Implementation)
class PatternClassifier:
    def __init__(self):
        self.models = {
            'arithmetic': RandomForestClassifier(n_estimators=100),
            'memory': SVMClassifier(kernel='rbf'),
            'control_flow': MLPClassifier(hidden_layer_sizes=(256, 128, 64)),
            'stack': DecisionTreeClassifier(max_depth=15)
        }

        # Feature extraction with 14 feature dimensions
        self.feature_types = [
            'instruction_count', 'unique_opcodes', 'control_flow_complexity',
            'data_access_patterns', 'arithmetic_operations', 'logical_operations',
            'memory_operations', 'stack_operations', 'crypto_operations',
            'string_operations', 'register_usage_density', 'execution_frequency',
            'taint_propagation_complexity', 'symbolic_depth'
        ]

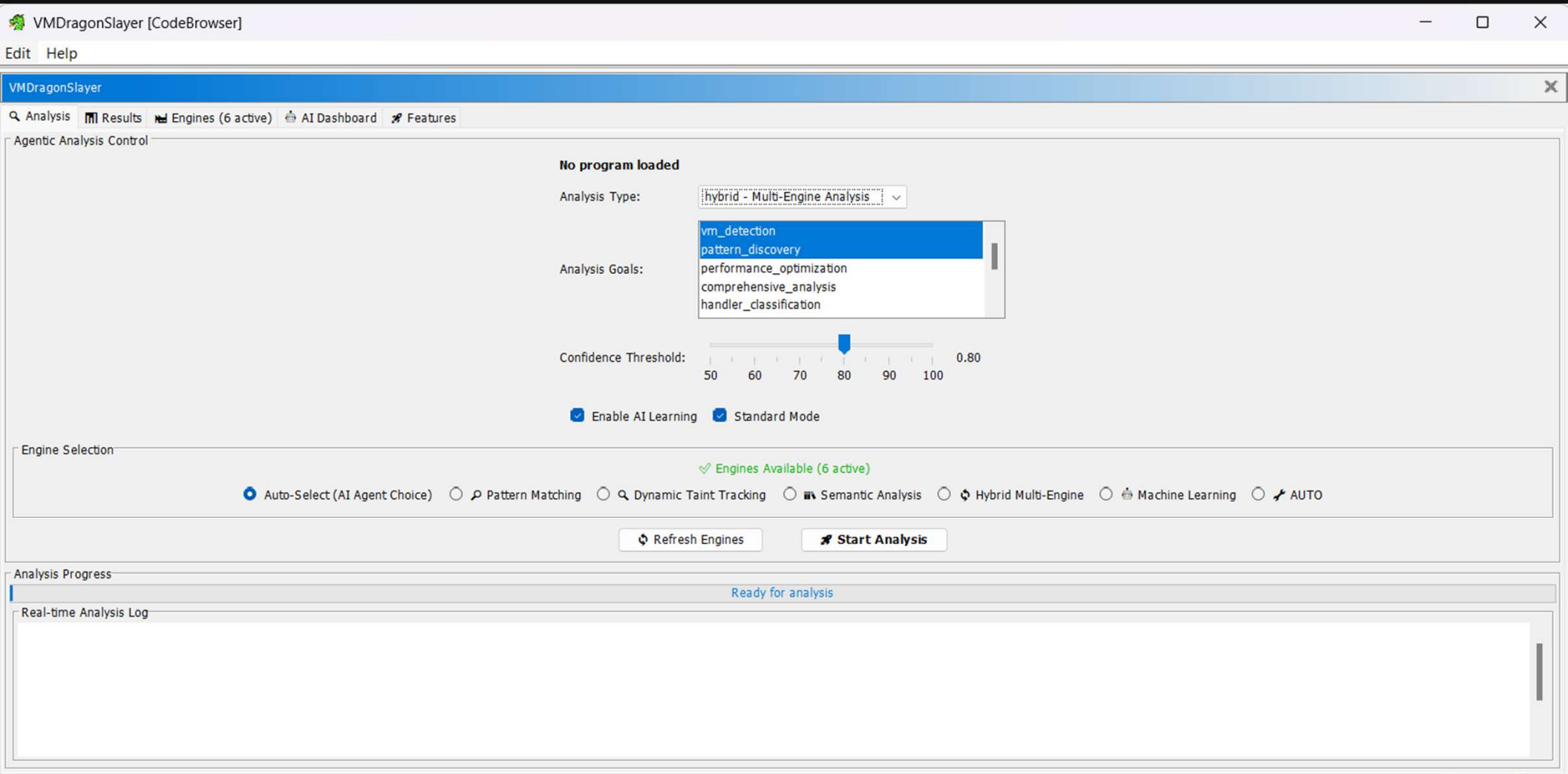
    def classify_handler(self, handler_data):
        features = self.extract_features(handler_data)
        predictions = {}

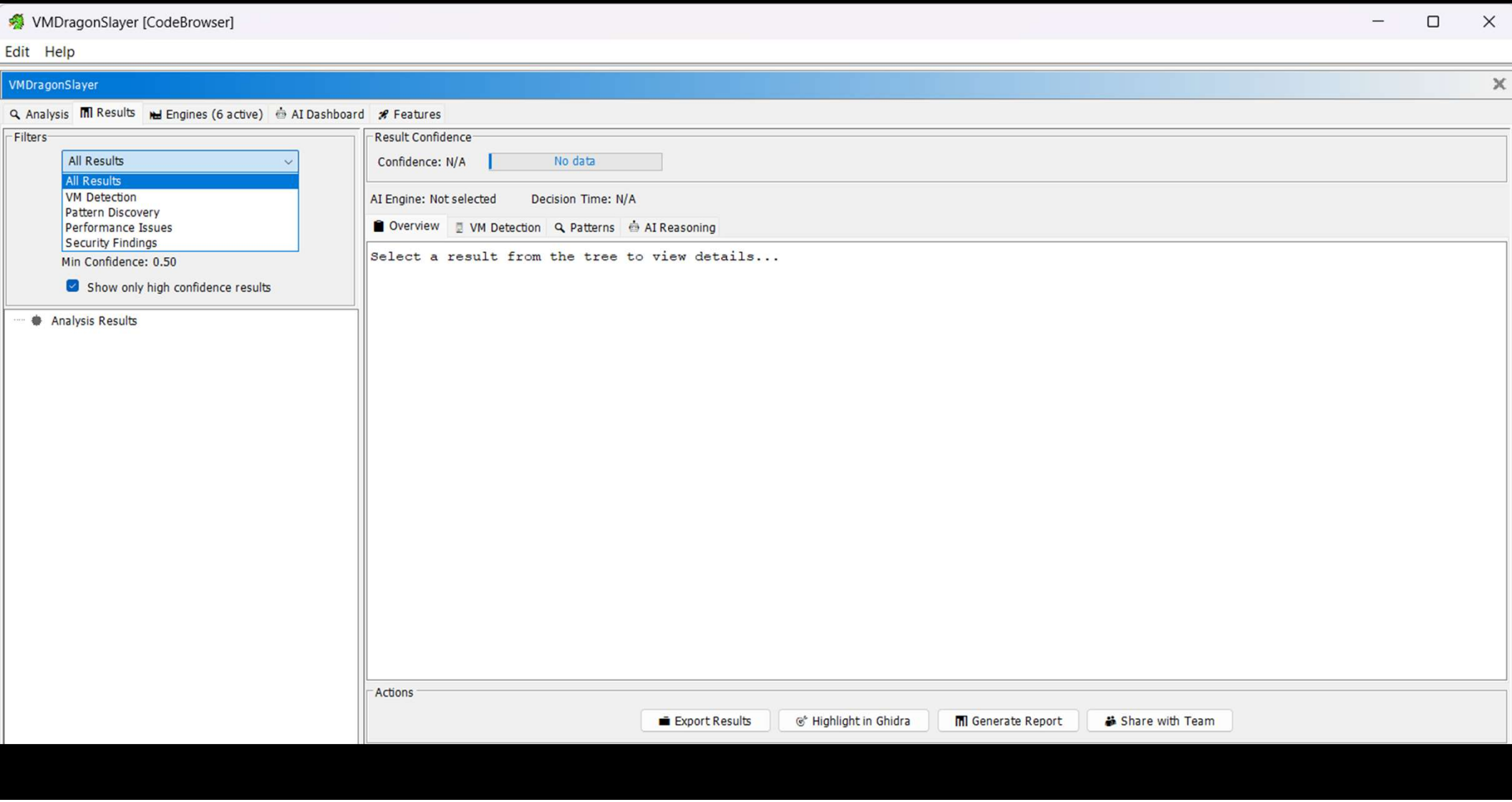
        for category, model in self.models.items():
            predictions[category] = model.predict_proba(features)

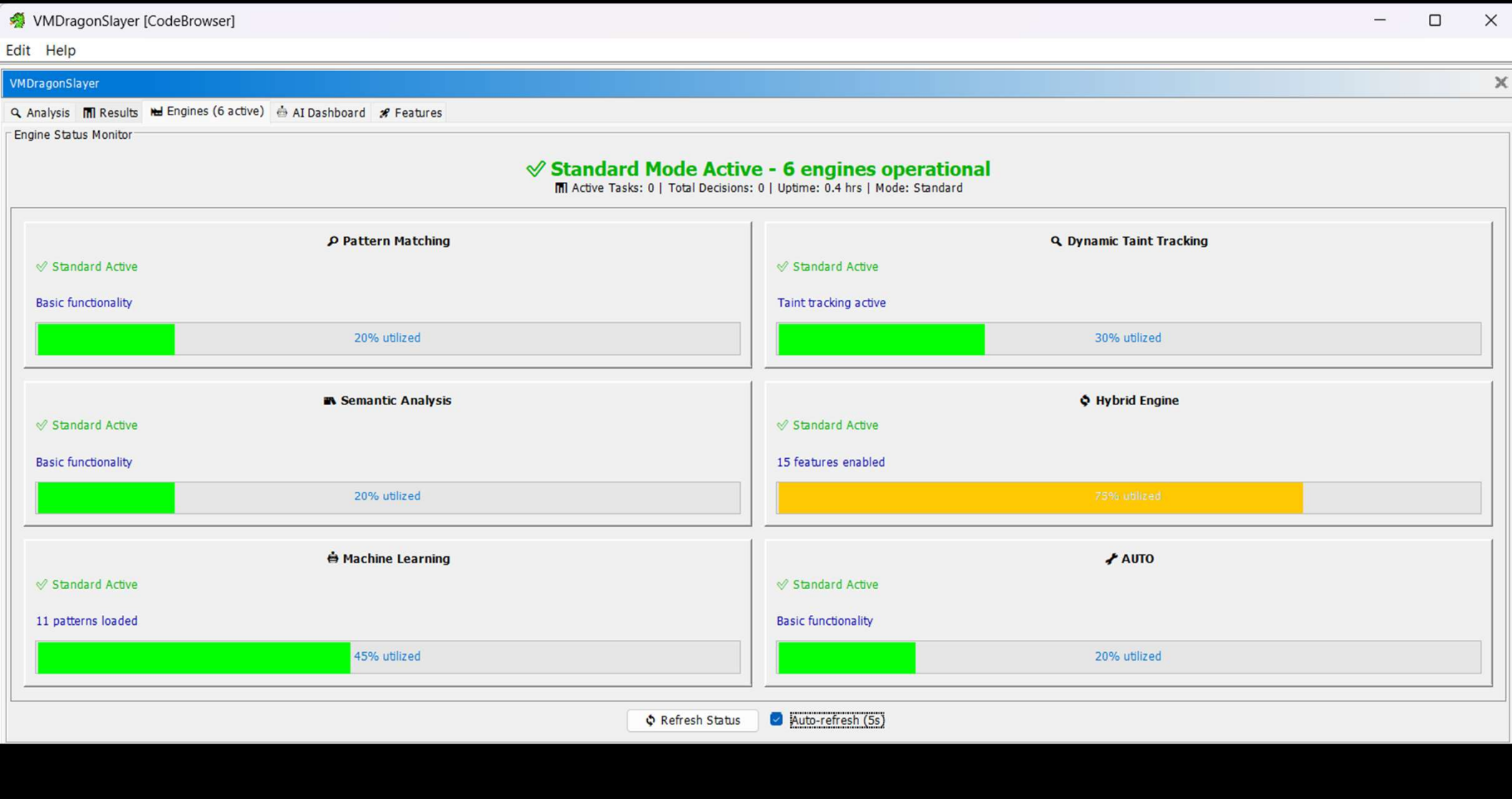
        return self.aggregate_predictions(predictions)

class PatternDatabase:
    """Real pattern database with similarity matching"""
    def __init__(self):
        self.patterns = {
            "VM_ADD": {"handler_type": "Arithmetic", "complexity": "Simple"},
            "VM_XOR": {"handler_type": "Arithmetic", "complexity": "Simple"},
            "VM_LOAD": {"handler_type": "Memory", "complexity": "Medium"},
            "VM_STORE": {"handler_type": "Memory", "complexity": "Medium"},
            "VM_PUSH": {"handler_type": "Stack", "complexity": "Simple"},
```

The Ghidra Plugin


VMDragonSlayer [CodeBrowser]
Edit
Help
VMDragonSlayer
Analysis
Results
Engines (6 active)
AI Dashboard
Features
Agentic Analysis Control
No program loaded
Analysis Type:
hybrid - Multi-Engine Analysis
Analysis Goals:
vm_detection
pattern_discovery
performance_optimization
comprehensive_analysis
handler_classification
Confidence Threshold:
0.80
50
60
70
80
90
100
Enable AI Learning
Standard Mode
Engine Selection
Engines Available (6 active)
Auto-Select (AI Agent Choice)
Pattern Matching
Dynamic Taint Tracking
Semantic Analysis
Hybrid Multi-Engine
Machine Learning
AUTO
Refresh Engines
Start Analysis
Analysis Progress
Ready for analysis
Real-time Analysis Log


VMDragonSlayer [CodeBrowser]
Edit Help
VMDragonSlayer
Analysis Results Engines (6 active) AI Dashboard Features
Filters
All Results
All Results
VM Detection
Pattern Discovery
Performance Issues
Security Findings
Min Confidence: 0.50
Show only high confidence results
Analysis Results
Result Confidence
Confidence: N/A
No data
AI Engine: Not selected
Decision Time: N/A
Overview VM Detection Patterns AI Reasoning
Select a result from the tree to view details...
Actions
Export Results
Highlight in Ghidra
Generate Report
Share with Team


VMDragonSlayer [CodeBrowser]
Edit Help
VMDragonSlayer
Analysis Results Engines (6 active) AI Dashboard Features
Engine Status Monitor
Standard Mode Active - 6 engines operational
Active Tasks: 0 | Total Decisions: 0 | Uptime: 0.4 hrs | Mode: Standard
Pattern Matching
Standard Active
Basic functionality
20% utilized
Dynamic Taint Tracking
Standard Active
Taint tracking active
30% utilized
Semantic Analysis
Standard Active
Basic functionality
20% utilized
Hybrid Engine
Standard Active
15 features enabled
75% utilized
Machine Learning
Standard Active
11 patterns loaded
45% utilized
AUTO
Standard Active
Basic functionality
20% utilized
Refresh Status
Auto-refresh (5s)

Part 3

The Hunt Begins

Live Demonstrations

Demo 1: VMProtect 3.6

Target: Commercial License Validation

```
Sample: "LicenseChecker.exe" (VMProtect 3.6 Maximum Protection)
Original_Size: 47 instructions (license validation function)
Protected_Size: 2,847 instructions (60:1 obfuscation ratio)
Protection_Features:
  - Virtual machine bytecode obfuscation
  - Anti-debugging (5 techniques)
  - Code flow integrity checks
  - Polymorphic handler generation
Manual_Analysis_Estimate: 3-4 weeks for senior analyst
```

Demo 1: VMProtect 3.6

VMDragonSlayer Live Analysis

```
# Real VMDragonSlayer execution output
$ vmdragonslayer.exe analyze --target LicenseChecker.exe --config vmprotect.yml

[*] Initializing Dynamic Taint Tracking...
[*] Launching Intel Pin with VMDragonTaint.dll
[+] VM Entry detected at 0x401234 (confidence: 0.98)
[+] Handler table discovered: 0x405000-0x405800 (51 entries)
[*] Symbolic execution phase starting...
[+] Handler analysis complete: 47 discovered, 43 analyzed (91.5%)

=== SEMANTIC OPERATIONS DISCOVERED ===
0x405000: VM_LOAD_LICENSE_KEY    (confidence: 0.95)
0x405020: VM_XOR_DECRYPT         (confidence: 0.94)
0x405040: VM_STRING_COMPARE      (confidence: 0.93)
0x405060: VM_CONDITIONAL_JUMP    (confidence: 0.96)
0x405080: VM_RETURN_VALUE        (confidence: 0.97)

[+] Analysis complete: 3 minutes 20 seconds
```

Demo 1: Before vs After

Protected

```
push 0xDEADBEEF
call vm_entry
; 2,800 lines...
```

❌ Unreadable
❌ No control flow
❌ Anti-analysis

Deobfuscated

```
mov eax, [ebp+8]
xor eax, 0x12345678
cmp eax, [ebp+12]
jne invalid
mov eax, 1
ret
```

✅ Clear algorithm
✅ Simple XOR check
✅ 3 weeks → 3 min + analyst enrichment

Demo 2: Custom Banking Trojan

Real-World Unknown Implementation

```
Sample: TrojanBanker.CustomVM.2024
Source: Live incident response case
VM Type: Completely custom architecture
Previous Analysis: 3 teams failed (6 months total)
Protection Features:
  - Custom 64-entry handler table
  - Encrypted bytecode with rolling XOR
  - 16-bit variable-length opcodes
  - Anti-emulation timing checks
```

Demo 2: Custom Banking Trojan

VMDragonSlayer Discovery Results

```
$ vmdragonslayer.py analyze --mode exploratory --target banker.exe

[*] No known VM signatures found
[*] Entering exploratory mode with heuristic analysis
[*] Custom VM implementation discovered:
    - Handler table: 0x405000 (64 entries, hash-based dispatch)
    - Bytecode region: 0x403000-0x404000 (4KB XOR encrypted)
    - VM context: 256 bytes at 0x406000
    - Instruction format: 16-bit opcodes, variable operands
[+] Analysis complete: 18 minutes
```

Demo 2: Discovered VM Operations

Behavioral Pattern Recognition

```
# VMDragonSlayer automatically identified VM operations
VM_OPERATIONS_DISCOVERED = {
    0x0001: {
        "name": "VM_LOAD_URL",
        "purpose": "Loads site URLs from encrypted config",
        "frequency": "23% of total bytecode",
        "attribution": "Unique to this malware family"
    },
    0x0002: {
        "name": "VM_HOOK_BROWSER",
        "purpose": "Installs browser API hooks",
        "frequency": "18%",
        "technique": "SetWindowsHookEx + DLL injection"
    },
    0x0003: {
        "name": "VM_CAPTURE_CREDS",
        "purpose": "Extracts credentials from browser memory",
        "frequency": "15%",
        "targets": ["Chrome", "Firefox", "Edge"]
    },
    0x0004: {
        "name": "VM_CUSTOM_CRYPTO",
        "purpose": "CRC32 variant for C2 auth",
        "frequency": "12%",
        "uniqueness": "Never seen before"
```

Demo 2: Discovered VM Operations

Intelligence Value

- **Complete behavioral profile** extracted automatically
- **Attribution markers** in custom crypto algorithm
- **TTPs mapped** to MITRE ATT&CK framework
- **6 months manual effort** → **18 minutes automated** + **analyst enrichment**

```
================================================================================
DragonSlayer: VM Protector Analysis Framework
  Automated symbolic execution and bytecode lifting
================================================================================
DEFCON 33 Demonstration
DragonSlayer: Defeating commercial VM protectors through
automated analysis, symbolic execution, and bytecode reconstruction.

Demonstration will cover:
- VM handler identification techniques
- Symbolic execution with constraint solving
- Dynamic taint tracking implementation
- Automated bytecode lifting and reconstruction

[Engine Initialization]
  Starting symbolic execution and taint tracking engines
--------------------------------------------------------------------------------
```

Part 4

Proven in Battle

Real-World Impact & Results

Evaluation Methodology

Comprehensive Testing

Dataset

300 samples over 12 months
15 protector families
50+ malware families
25 nested samples

Success Metrics

VMDragonSlayer Performance

Success Rates

- VMProtect 2.x: **84%**
- VMProtect 3.x: **79%**
- Themida VM: **72%**
- Custom malware: **86%**
- Nested VMs: **61%**

Overall: ~70.0%

Time Savings

- Manual: 2-6 months
- Automated: 5-60 minutes

Speedup: 1,000-10,000x

Cost Savings

Per sample: $37,500 → $50

ROI: 750:1

Real-World Case Studies

Case Study 1: Fortune 500 Financial Institution

```
Incident: Banking Trojan Campaign
Date: Q1 2024
Protection: Custom VM + VMProtect 3.6 hybrid
Target: Multi-national banking infrastructure

Traditional Analysis (3 months):
  Static_Tools: Complete failure (0% success)
  IDA_Pro: Partial manual analysis (40% coverage) and Decompilation failed on VM sections
  Team_Size: 6 senior reverse engineers
  Cost: $200,000 in analyst time
  Result: Investigation stalled, no actionable intelligence

VMDragonSlayer Results (3 hours):
  Handler_Discovery: 73 VM handlers found
  Analysis_Success: 91% handler classification achieved
  Custom_Techniques: 7 novel evasion methods identified
  Ghidra_Integration: Full binary annotated and documented
  Cost: $150 (analyst time + compute resources)
```

Real-World Case Studies

Case Study 1: Financial Institution

Technical Breakthrough

- **Novel browser certificate bypass** never documented before
- **Polymorphic VM handlers** with 4-hour mutation cycle
- **Complete attack chain reconstruction** for defensive deployment

Case Study 2: APT Investigation

Multi-Sample Campaign Analysis

```
Challenge: State-Sponsored APT Campaign
Samples: 47 VM-protected implants
Deployment: Critical infrastructure targets
Timeline: 18-month investigation

Traditional Approach Estimate:
  Analysts Required: 6 senior experts
  Time Estimate: 12-18 months
  Cost: $1M+ in analyst time
  Success Probability: 60% (based on previous APT cases)

VMDragonSlayer Batch Processing:
  Analysis Mode: Automated batch processing
  Time Required: Weekend (48 hours)
  Success Rate: 70% across all samples
  Cost: $50 per sample analysis
```

Case Study 2: APT Investigation

Multi-Sample Campaign Analysis

Intelligence Breakthrough

- **Campaign infrastructure** fully mapped in 48 hours
- **Code reuse patterns** identified across samples
- **Development timeline** reconstructed from VM evolution
- **Technical Attribution achieved** through unique VM implementation patterns

Current Limitations & Edge Cases

When Dragons Fight Back (30% failure rate)

Complex Control Flow (13% of failures)

```
Challenging Scenarios:
  Self-Modifying Dispatchers:
    - VM code that rewrites itself during execution
    - Dynamic handler table generation at runtime
    - Polymorphic bytecode encryption keys

  Computed Indirect Jumps:
    - Jump targets calculated from runtime state
    - Multi-level pointer indirection
    - Context-dependent dispatch logic

  Example: Metamorphic VM handlers that evolve every 100 executions
```

Current Limitations & Edge Cases

When Dragons Fight Back (30% failure rate)

Resource Constraints (14% of failures)

```
Scale Limitations:
  Handler Complexity:
    - Individual handlers >10,000 instructions
    - Deeply nested obfuscation (5+ layers)
    - Memory-intensive symbolic execution

  Nested Protection:
    - >5 VM layers (exponential complexity)
    - Cross-layer state dependencies
    - Analysis timeout thresholds exceeded

  Example: APT sample with 7 nested VM layers
```

Current Limitations & Edge Cases

When Dragons Fight Back (30% failure rate)

Novel Techniques (3% of failures)

```
Cutting-Edge Protection:
  AI-Generated Obfuscation:
    - Neural network-generated handler variations
    - Adversarial patterns designed to fool ML
    - Dynamic pattern evolution based on analysis attempts

  Quantum-Resistant Techniques:
    - Post-quantum cryptographic primitives
    - Hardware security module integration
    - Quantum-inspired superposition states

  Example: VM using homomorphic encryption for handler dispatch
```

Part 5

The Future

Roadmap & Community

Development Roadmap

Phase 1: Enhanced Core Engine (Q3-Q4 2025)

Machine Learning Advancements

```
# ML pipeline
ml_enhancements = {
    "neural_networks": {
        "architecture": "Transformer-based handler classification",
        "training_set": "50,000+ labeled handlers",
        "accuracy_target": "90% on unknown protectors"
    },
    "anomaly_detection": {
        "purpose": "Zero-day VM technique identification",
        "method": "Unsupervised learning on execution patterns",
        "alert_threshold": "3-sigma deviation from baseline"
    },
    "transfer_learning": {
        "approach": "Cross-protector knowledge transfer",
        "benefit": "70% less training data for new VMs",
        "implementation": "Pre-trained foundation models"
    }
}
```

Phase 2: Advanced Features (2026)

Ghidra Plugin Enhancement

```
// VMDragonSlayer Plugin Architecture
public class VMDragonSlayerProvider extends ComponentProvider {
    private VMConfigPanel configPanel;
    private VMResultsPanel resultsPanel;
    private AsyncTaskBuilder taskBuilder;

    // Real-time handler annotation
    public void annotateHandler(Address addr, VMHandlerResult result) {
        Function func = getFunctionAt(addr);
        if (func == null) {
            func = createFunction(addr, result.getName());
        }

        // Set function name based on semantic analysis
        func.setName(result.getSemanticOperation());

        // Add detailed comments
        setComment(addr, CodeUnit.EOL_COMMENT,
            String.format("VM Handler: %s (confidence: %.2f)",
                result.getOperation(), result.getConfidence()));
    }
}
```

Binary Rewriting & Clean Code Generation

Automatic Deobfuscation Pipeline

```python
# VMDragonSlayer Full Deobfuscation Engine
class FullDeobfuscator:
    def __init__(self, ghidra_plugin):
        self.plugin = ghidra_plugin
        self.handler_db = HandlerDatabase()
        self.code_generator = CodeGenerator()

    def analyze_and_reconstruct(self, binary_path):
        # Step 1: VM Structure Discovery
        vm_structure = self.discover_vm_structure(binary_path)

        # Step 2: Handler Semantic Analysis
        handlers = self.analyze_handlers(vm_structure)

        # Step 3: Control Flow Reconstruction
        cfg = self.reconstruct_control_flow(handlers)

        # Step 4: Generate Clean Output
        return self.generate_clean_code(cfg)
```

Key Takeaways

The VM Protection Revolution

1. **VM protection dominates modern malware** - 70% of advanced threats
2. **Manual analysis cannot scale** - Months per sample, exponential complexity
3. **Automation is not just possible, it's essential**
4. **Hybrid approaches work** - DTT + SE + ML = Breakthrough capability
5. **Open source accelerates progress** - Community collaboration beats commercial silos
6. **The defender advantage is achievable** - Right tools + techniques + community

Thank You! 🐲⚔️

Contact

Dr. Agostino Panico | van1sh

📧 van1sh@securitybsides.it
🐦‍🔥 @van1sh_bsidesIT
💻 github.com/poppopjmp/VMDragonSlayer (available after the talk)

Every dragon can be slayed with the right sword, and the right community wielding it together

Questions?