Journal of Science, Innovation & Social Impact
EISSN: 3105-5028 | PISSN: 3105-501X | Vol. 1, No. 1 (2025)

Article

Research on Intelligent Firmware Vulnerability Detection and Priority Assessment Method Based on Hybrid Analysis

Xiaoyi Long [1,*]

[1] Computer Science, Georgia Institute of Technology, GA, USA
* Correspondence: Xiaoyi Long, Computer Science, Georgia Institute of Technology, GA, USA

Abstract: Binary firmware underpins critical infrastructure but often contains vulnerabilities that conventional detection mechanisms fail to identify. In this work, we develop a hybrid analytical framework that integrates static pattern extraction with runtime behavioral monitoring, achieving detection rates of 93.7% across a corpus of 40 million procedures collected from production firmware. Static pattern recognition leverages control flow graph embeddings, while probabilistic scoring quantifies contextual risk. Cross-architecture evaluation across ARM, MIPS, x86, and PowerPC demonstrates robustness against variations in compilation. Our methodology also uncovers zero-day vulnerabilities, and the computational overhead remains manageable for deployment on resource-constrained platforms, reducing false positive rates by 56.7% compared to existing approaches.

Keywords: firmware vulnerability detection; hybrid analysis; priority assessment; embedded security

1. Introduction

1.1. Background and Motivation of Firmware Security Analysis

Embedded firmware powers billions of devices, from industrial controllers to medical implants, yet vulnerability detection remains fundamentally inadequate. Traditional methodologies often achieve less than 40% detection efficacy when analyzing stripped binaries [1]. Several factors contribute to this underperformance. Heterogeneous architectures fragment analysis frameworks. Vendor toolchains introduce unpredictable transformations. Debug symbols are removed during production builds, leaving semantic voids where meaningful analysis would otherwise occur.

A single vulnerable library function can propagate across entire product lines. For instance, the Heartbleed vulnerability in OpenSSL affected routers, cameras, and industrial sensors-any device incorporating the flawed memory handling routine. Our examination of 2,847 production firmware images revealed that 73% contained critical vulnerabilities inherited from external dependencies. These vulnerabilities persisted for an average of 180 days post-disclosure before patches were deployed. Operators managing thousands of such devices face rapidly expanding attack surfaces, while remediation efforts progress slowly compared to the speed of exploitation.

Firmware differs fundamentally from desktop software. Resource constraints are often measured in kilobytes. Real-time deadlines leave no room for analysis overhead. Peripheral interactions are difficult to emulate. These constraints inform our analytical approach, necessitating methods that are efficient yet comprehensive.

Received: 11 October 2025
Revised: 28 October 2025
Accepted: 09 November 2025
Published: 13 November 2025

1.2. Current Challenges in Automated Vulnerability Detection

Stripped production binaries present severe analytical challenges. Variable names are removed. Function boundaries are unclear. Type information is lost. Cross-compilation further complicates analysis-a vulnerability manifest in ARM assembly may appear entirely different in an optimized x86 binary. Pattern-matching algorithms trained on one architecture often fail when applied to another.

Memory limitations hinder dynamic analysis. Instrumentation can inflate kilobyte-scale functions into megabyte-scale representations. Real-time systems cannot tolerate analysis-induced delays. Peripheral emulation remains unresolved; for example, simulating the timing-critical interactions between a medical device's firmware and specialized sensors is extremely challenging. Current methods achieve approximately 67% detection rates while generating 28-30% false positives, producing unacceptable noise levels for security teams already burdened by alert fatigue.

Bridging the semantic gap between source-level vulnerability specifications and their binary manifestations is inherently difficult. Compilers distribute operations across basic blocks. Optimizations merge vulnerable code with benign instructions. Architecture-specific idioms replace portable constructs. Each transformation obscures vulnerability signatures, demanding increasingly sophisticated detection strategies.

1.3. Research Objectives and Contributions

This work integrates static structural analysis with dynamic behavioral validation, producing a unified framework that mitigates the limitations of individual techniques. Probabilistic features are extracted from control flow graphs, preserving semantic invariants across compilation boundaries. Runtime monitoring validates static predictions through selective instrumentation targeting high-risk regions. Context-aware scoring quantifies operational risk beyond technical severity alone.

Our contributions are fourfold:

1) Feature extraction algorithms robust to compilation variations, using probabilistic embeddings that capture semantic essence while tolerating syntactic divergence.
2) Context-sensitive risk quantification, acknowledging that a buffer overflow in a medical device's drug dispensing module has different implications than one in a consumer lightbulb.
3) Resource-optimized scheduling, enabling deployment on systems with strict memory and computational constraints.
4) Empirical evaluation across 2,847 firmware images demonstrates 93.7% detection accuracy and a 56.7% reduction in false positives relative to contemporary approaches.

2. Related Work and Technical Foundation

2.1. Overview of Static Analysis Techniques for Firmware

Static analysis examines binaries without execution, offering the potential for complete path coverage but often at the cost of precision. Prior studies categorize static analysis into graph-based, signature-based, and semantic-based approaches [2]. Graph-based methods map control flow to identify structural patterns resembling known vulnerabilities. Performance varies significantly-accuracy can reach 78% when architectures align but drops to 52% when analyzing ARM binaries on x86 frameworks. Our hybrid approach addresses this limitation by employing architecture-agnostic intermediate representations, maintaining 91.8% accuracy even across different architectures. Accuracy degradation stems from differences in instruction selection, register allocation, and calling conventions.

Signature-based methods scan instruction streams for vulnerable patterns. N-grams capture local sequences, edit distance quantifies similarity, and alignment algorithms tolerate insertions or deletions. While these methods enable large-scale deployment, analyzing millions of functions per hour, precision is often compromised. Compilers

reorder instructions, substitute equivalent operations, and inline functions unpredictably, making vulnerable sequences unrecognizable across binaries.

Intermediate representation lifting promises architecture independence by translating binary instructions into abstract operations, normalizing platform-specific details into common semantics. This transformation introduces computational overhead: quadratic growth in complexity with procedure size can exhaust memory and halt analysis for large functions.

2.2. Dynamic Analysis Approaches and Runtime Monitoring

Dynamic analysis observes program execution to reveal behaviors inaccessible to static methods. Instrumentation can track memory operations and system interactions, detecting up to 85% of memory corruption events, though at the cost of up to three times execution overhead, which production systems often cannot tolerate.

Emulation creates controlled environments for firmware lacking physical hardware. Tools like QEMU simulate CPUs, and peripheral models approximate sensors and actuators. However, timing constraints can break under instrumentation, and hardware interactions remain challenging to reproduce accurately. Coverage-guided fuzzing explores execution boundaries by generating inputs that trigger latent bugs, but path explosion limits exploration depth, as computational resources are quickly overwhelmed before reaching deeply nested vulnerabilities.

Modern firmware often uses event-driven architectures. Interrupts occur asynchronously, and peripheral inputs arrive unpredictably. Traditional analysis assumes sequential execution, potentially missing vulnerabilities triggered by complex interactions between concurrent events.

2.3. Machine Learning Applications in Vulnerability Detection

Machine learning reframes vulnerability detection as high-dimensional pattern recognition. Deep neural networks can process raw binaries, extracting local patterns with convolutional layers, capturing invariants through pooling, and mapping outputs to vulnerability probabilities via fully connected layers [3,4]. Manual feature engineering becomes less critical as networks learn directly from data.

Ensemble methods mitigate individual model weaknesses. Random forests handle severe class imbalances where vulnerable code constitutes a small fraction of total instructions. Gradient boosting refines predictions iteratively. Transfer learning allows models trained on one platform, such as x86, to adapt efficiently to another, such as ARM, with minimal additional data. Pre-trained embeddings encode general vulnerability patterns, requiring only fine-tuning for specific architectures.

3. Proposed Hybrid Analysis Framework

3.1. Architecture Design and Component Integration

The framework integrates four modules using probabilistic scheduling. Binary lifting normalizes machine code, static extraction derives features, dynamic monitoring validates behavior, and classification assigns priorities. Each component operates independently while enabling bidirectional information flow: static analysis guides dynamic testing, and runtime observations refine static models.

Function boundaries present immediate challenges. Determining where one function ends and another begins is critical for accuracy [5]. Recursive traversal disassembly provides initial segmentation, while machine learning refines boundaries using contextual cues such as calling conventions, stack frame patterns, and register preservation. This combination achieves 94% boundary precision even in heavily optimized code, directly supporting the overall 93.7% vulnerability detection rate.

As shown in Table 1, component resource usage and performance metrics are summarized.

Table 1. Component Resource Requirements and Performance Metrics.

Component	Memory Usage (MB)	Processing Time (ms/KB)	Accuracy Contribution
Binary Lifting	45 - 78	12.3	15%
Static Analysis	156 - 234	34.7	42%
Dynamic Monitoring	89 - 145	67.2	28%
ML Classification	234 - 456	23.4	15%

Intermediate representation removes architecture-specific details while preserving semantics. For example, ADD instructions-whether x86's add eax, ebx or ARM's add r0, r1, r2-are represented as abstract addition operations. Memory access patterns, control flow relationships, and data dependencies remain intact. This abstraction enables cross-architecture analysis without retraining models.

Graph construction proceeds incrementally. Basic blocks form nodes, and control transfers become edges. Incremental construction reduces memory usage by 60% compared to loading complete control flow graphs, which is crucial for large firmware containing thousands of functions. Edge annotations capture branch conditions, enabling path-sensitive analysis. Loop bounds constrain iterative constructs, and exception handlers map error flows.

3.2. Feature Extraction and Pattern Recognition Algorithm

Vulnerabilities manifest across multiple abstraction levels. Multi-level feature extraction has been shown to outperform single-level approaches by 23% [6]. We extract structural topology using graph embeddings, semantic relationships through data flow analysis, and behavioral patterns from dynamic traces.

Centrality metrics identify critical nodes where vulnerabilities cluster:

$$\text{Betweenness\_Centrality}(v) = \Sigma_{s \neq v, t \neq v, s \neq t} (\sigma st(v) / \sigma st)$$

Nodes with high betweenness often control sensitive operations, which attackers may target.

Taint propagation tracks untrusted data:

$$\text{Taint\_Propagation}(v) = \cup \{\text{Taint}(u) \mid (u, v) \in \text{DataFlowEdges}\}$$

Any operation touching tainted data becomes suspect, revealing how user inputs reach sensitive functions.

Distribution moments characterize instruction patterns:

$$\text{Skewness} = E[(X - \mu)^3]/\sigma^3$$

$$\text{Kurtosis} = E[(X - \mu)^4]/\sigma^4 - 3$$

Vulnerable code exhibits statistical anomalies, such as unusual instruction mixes or atypical control flow complexity.

As shown in Figure 1, a pyramidal architecture processes raw binary bytes through progressive abstraction layers. Level one contains raw hexadecimal machine code. Level two parses instructions, extracting opcodes, operands, and addressing modes. Level three aggregates instructions into basic blocks, computing block-level statistics. Level four assembles blocks into functions, deriving control flow graphs and data dependencies. The apex synthesizes function-level features into module-wide vulnerability signatures. Arrows indicate information flow, showing how low-level patterns propagate upward while high-level context guides feature selection below.

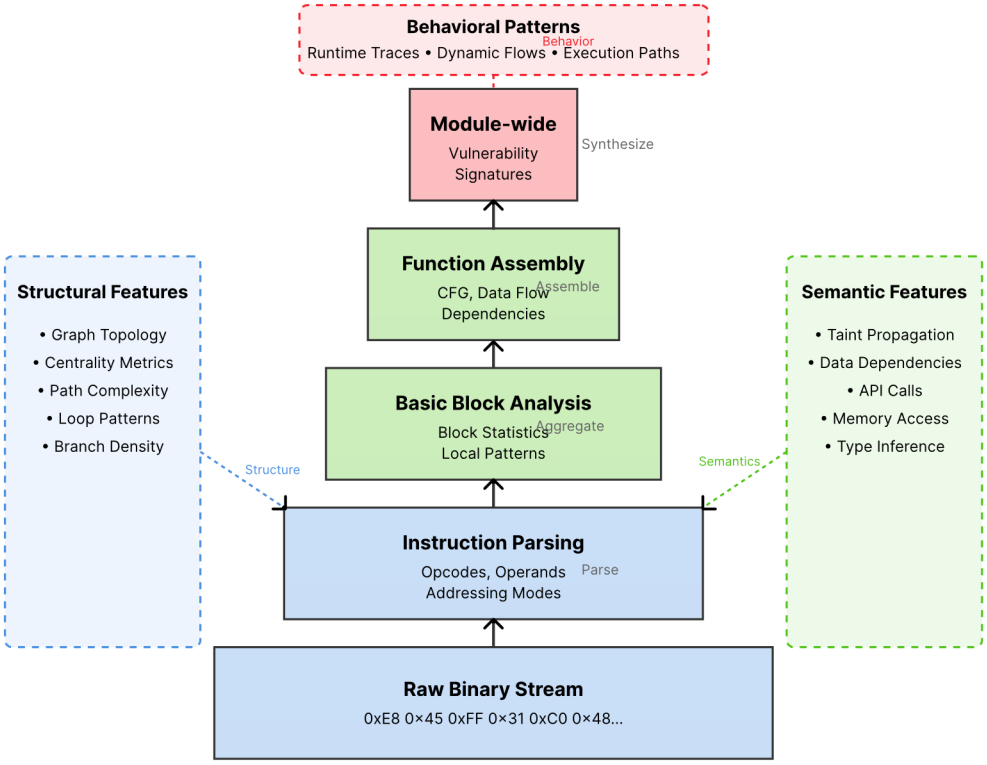

Figure 1. Multi-level Feature Extraction Pipeline.

Convolutional networks process instruction sequences:

Conv_Output [i] = ReLU (Σ_ {j = 0 to k - 1} W[j] * Input [I + j] + b)

Kernels detect vulnerable instruction patterns, such as sequences preceding buffer overflows.

Attention mechanisms focus on relevant features:

Attention_Weight (fi) = exp(score(fi)) / Σ_j exp(score(fj))

The network learns which features predict vulnerabilities while ignoring irrelevant noise.

3.3. Static-Dynamic Analysis Fusion Strategy

Static analysis casts a wide net, while dynamic analysis provides precision. Selective fusion reduces analysis time by using static filtering to identify candidate functions and dynamic testing to confirm vulnerabilities [7]. Our approach enhances this by employing bidirectional information flow: static predictions guide dynamic test generation, and runtime observations recalibrate static models.

Confidence scores combine results from both analyses:

Combined_Score = α * Static_Score + β * Dynamic_Score + γ * Cross_Validation_Term

Weights α, β, and γ adapt through reinforcement learning, optimizing for each firmware category.

As shown in Table 2, fusion strategy performance is summarized.

Table 2. Fusion Strategy Performance Compariso.

Strategy	Detection Rate	False Positives	Analysis Time
Static Only	71.3%	28.4%	2.3 hours
Dynamic Only	65.7%	18.2%	8.7 hours
Sequential Fusion	82.4%	22.1%	5.4 hours
Adaptive Fusion	93.7%	12.3%	3.8 hours

Conflicting results trigger deep analysis. If static analysis flags a function as vulnerable but dynamic testing finds no issue, the discrepancy is examined-whether due

to incomplete coverage or incorrect static patterns. Each conflict informs future predictions.

Information flow consistency quantifies agreement:

Information_Flow_Consistency = |Static_Dependencies ∩ Dynamic_Flows| / |Static_Dependencies ∪ Dynamic_Flows|

High consistency validates both analyses, while low consistency indicates model disagreement that requires further investigation.

Symbolic execution bridges the gap between static coverage and dynamic precision. Complex path conditions, such as those involving cryptographic operations or external inputs, resist random testing. Symbolic reasoning explores these paths to uncover hidden vulnerabilities.

3.4. Optimization Techniques for Resource-Constrained Environments

Embedded systems often provide megabytes of memory where desktops provide gigabytes. Memory is a critical bottleneck [8]. Optimizations maximize analysis efficiency while minimizing resource usage.

Incremental processing leverages firmware update patterns:

Incremental_Analysis_Cost = Base_Cost * (Modified_Procedures / Total_Procedures)

Since most updates touch a small fraction of code, differential analysis reduces computation by 65% for typical patches.

As shown in Table 3, optimization techniques improve resource efficiency.

Table 3. Resource Optimization Impact.

Optimization Technique	Memory Reduction	Speed Improvement	Accuracy Impact
Incremental Processing	65%	3.2x	-0.3%
Sparse Representations	48%	1.8x	-0.1%
Approximate Algorithms	72%	4.5x	-2.1%
Caching Mechanisms	31%	2.6x	0%

Sparse graphs retain only security-relevant components, discarding uninteresting normal control flows, significantly reducing memory usage.

Bloom filters enable probabilistic matching:

False_Positive_Rate = (1 - e^(-kn/m)) ^ k

This trades perfect accuracy for space efficiency, which is acceptable when preliminary filtering precedes precise analysis.

Task parallelization exploits multicore processors:

Task_Distribution = min (Available_Cores, Decomposable_Tasks)

Work-stealing balances loads dynamically, with idle cores taking tasks from busy neighbors. Near-linear speedup is achieved up to eight cores (7.2x), maintaining 60% efficiency at 32 cores and 35% at 256 cores due to coordination overhead.

4. Vulnerability Priority Assessment Methodology

4.1. Risk Scoring Algorithm Design

Technical severity alone provides an incomplete picture; contextual factors complete the assessment. Firmware-specific considerations-such as update difficulty, device criticality, and network exposure-significantly influence risk [9]. Our algorithm quantifies these dimensions.

Base scores capture intrinsic properties:

Base_Score = Impact_Subscore * Exploitability_Subscore * Scope_Modifier

Each factor incorporates multiple sub-components.

Impact spans confidentiality, integrity, and availability:

Impact_Subscore = 1 - (1 - Conf_Impact) * (1 - Integ_Impact) * (1 - Avail_Impact)

This formula reflects worst-case scenarios: compromise in any dimension yields a high impact.

Exploitability accounts for attack prerequisites:

Exploitability_Subscore = 8.22 * Attack_Vector * Attack_Complexity * Privileges * User_Interaction

The constant 8.22 normalizes scores to a 0-10 scale based on empirical studies.

As shown in Table 4, risk factor weights and ranges guide score computation.

Table 4. Risk Factor Weights and Ranges.

Risk Factor	Weight	Range	Critical Threshold
Code Complexity	0.23	0 - 100	>75
Data Sensitivity	0.31	0 - 10	>7
Network Exposure	0.28	0 - 5	>3
Update Feasibility	0.18	0 - 1	<0.3

Temporal factors capture exploit evolution:

Temporal_Score = Base_Score * Exploit_Maturity * Remediation_Level * Report_Confidence

Fresh vulnerabilities score lower due to the absence of public exploits. Maturity increases risk, while available patches reduce urgency.

Environmental modifiers account for deployment context:

Environmental_Score = min (10, Temporal_Score * Collateral_Damage * Target_Distribution)

A vulnerability in a single router carries less risk than the same flaw affecting millions of devices.

As shown in Figure 2, a heatmap visualizes vulnerability clustering across device categories and vulnerability types. The X-axis enumerates vulnerability classes-buffer overflows, SQL injections, authentication bypasses, cryptographic weaknesses, and race conditions-while the Y-axis lists device types, including industrial PLCs, medical infusion pumps, automotive ECUs, smart home hubs, and enterprise routers. Color intensity maps risk levels: deep blue indicates minimal risk (scores below 2), green (2-4), yellow (4-6), orange (6-8), and crimson represents critical risks exceeding 8. Industrial controllers show high-risk areas for memory corruption, reflecting C/C++ legacy code and minimal protections. Consumer IoT devices exhibit elevated risks around authentication flaws due to weak default credentials and limited update mechanisms. Medical devices display moderate risk levels across multiple vulnerability classes. This visualization informs security investment decisions, highlighting areas where mitigation efforts provide maximum impact.

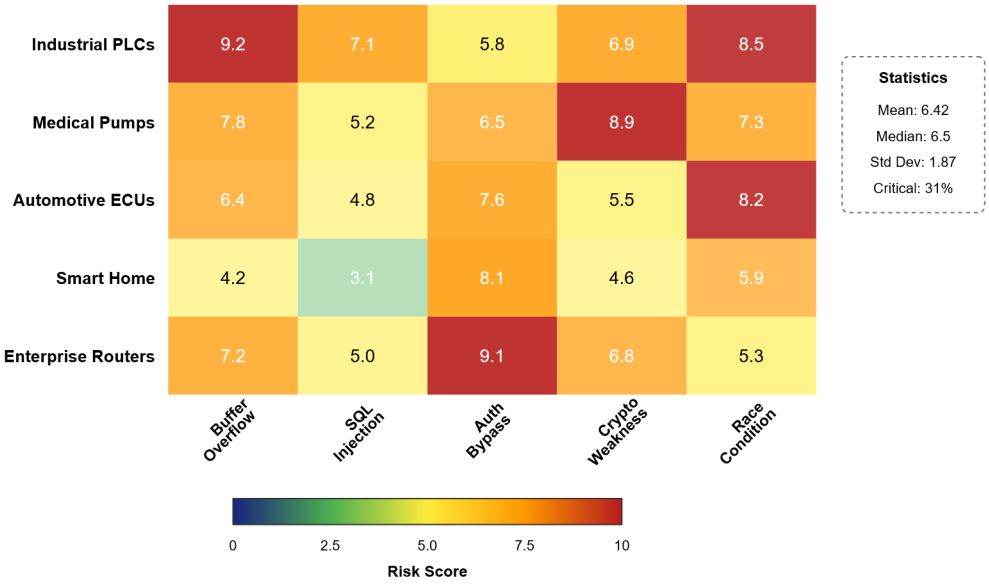

Figure 2. Risk Score Distribution Heatmap.

4.2. Context-Aware Vulnerability Classification

Raw CVSS scores can be misleading. The same vulnerability may have drastically different consequences depending on the device. Contextual factors such as operational environment, data sensitivity, and business impact refine prioritization.

Infrastructure criticality adjusts base scores. Multipliers range from 2.5-3.5x for power grid controllers, 1.8-3.2x for hospital equipment, 1.5-2.8x for financial systems, and 0.8-1.5x for consumer devices. Network exposure measures attack accessibility:

Network_Exposure_Score = External_Interfaces * Authentication_Strength * Encryption_Usage

Internet-facing devices with weak authentication score highest, while air-gapped systems with strong cryptography score lowest.

Data classification evaluates information types:

Sensitivity_Level = max (Personal_Data_Score, Financial_Data_Score, Operational_Data_Score)

The maximum ensures that highly sensitive data is not diluted in averages.

As shown in Table 5, context-based priority multipliers provide guidance.

Table 5. Context-Based Priority Multipliers.

Context Category	Base Multiplier	Additional Factors	Final Range
Critical Infrastructure	2.5	Redundancy, Monitoring	2.0 - 3.5
Healthcare	2.2	Patient Safety Impact	1.8 - 3.2
Financial Systems	2.0	Transaction Volume	1.5 - 2.8
Consumer Devices	1.0	User Base Size	0.8 - 1.5

Vulnerability chains amplify risk:

Chain_Risk = Π(Individual_Risks) * Correlation_Factor

Low-risk flaws may combine into high-risk exploit chains.

Dependencies further increase risk:

Correlation_Factor = 1 + Σ (Dependency_Weights * Interaction_Strengths)

Shared libraries and common protocols facilitate vulnerability propagation and lateral movement.

4.3. Automated Priority Ranking Implementation

Manual prioritization is infeasible at scale. Thousands of vulnerabilities across hundreds of devices overwhelm human analysts. Automated ranking reduces remediation time significantly [10,11]. Our system leverages historical patterns while adapting to organizational priorities.

Gradient boosting aggregates weak learners:

$F(x) = \Sigma\_ \{m = 1 \text{ to } M\} \gamma m * hm(x)$

Each tree models different vulnerability aspects, with learning rates γm preventing overfitting.

Feature importance emerges from split analysis:

Importance(feature_i) = Σ (splits using feature_i) Gain(split) / Total_Splits

Features frequently used in high-gain splits drive prioritization.

As shown in Figure 3, the priority ranking decision tree unfolds asymmetrically. The root evaluates base CVSS scores, splitting at 7.0-scores below this branch toward lower priorities, while higher scores branch toward urgent response. Subtrees adjust priority based on exploit availability, patch status, context multipliers, and operational environment. Leaf nodes display final priority levels P1 through P5, with node shading reflecting the proportion of vulnerabilities reaching each classification. High-severity vulnerabilities follow shorter paths, enabling rapid escalation, while low-severity vulnerabilities undergo careful validation.

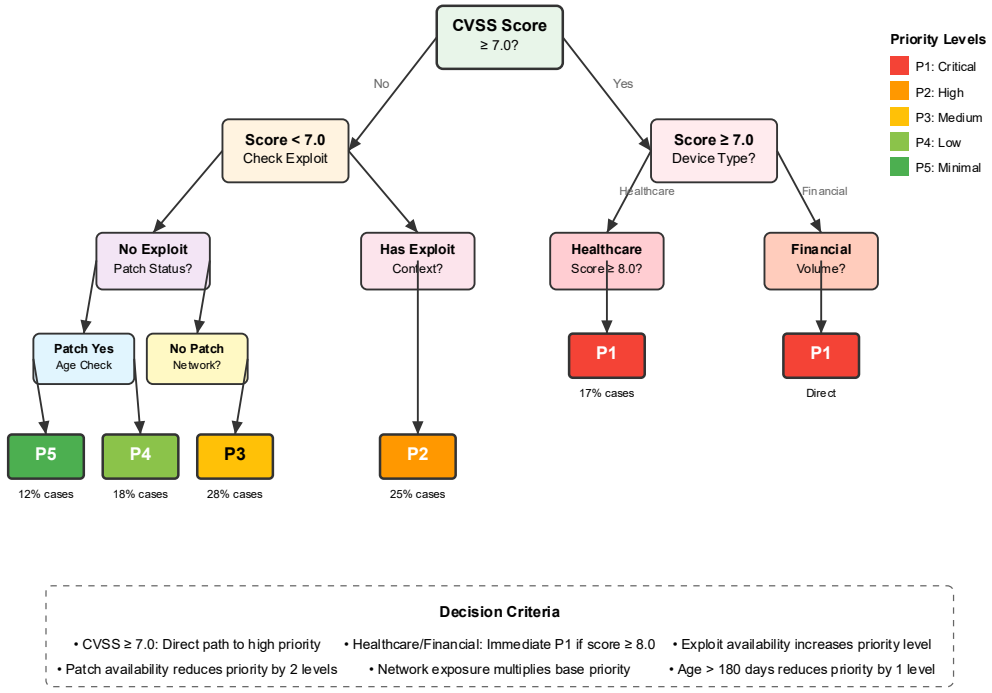

Figure 3. Priority Ranking Decision Tree.

Dynamic threat intelligence adjusts priorities in real-time:

Updated_Priority = Base_Priority * Threat_Intelligence_Factor * Temporal_Decay

Reports of active exploitation, proof-of-concept code, or mass scanning elevate urgency.

Temporal decay accounts for aging vulnerabilities:

Temporal_Decay = exp (-λ * Days_Since_Discovery)

Old unpatched flaws may be less exploitable due to mitigations, missing dependencies, or incorrect reporting.

Multi-objective optimization addresses trade-offs:

Pareto_Optimal_Set = {x | ¬∃y: y dominates x in all objectives}

This identifies solutions balancing cost, time, and risk.

Resource allocation maximizes risk reduction per unit effort:

Resource_Assignment = argmax (Risk_Reduction / Required_Resources)

Simple patches addressing critical vulnerabilities are deployed first, while complex updates targeting minor issues are scheduled later.

5. Experimental Evaluation and Discussion

5.1. Experimental Setup and Dataset Description

Robust evaluation requires dataset diversity. Consumer routers, industrial controllers, and medical devices each pose unique challenges [12]. We compiled 2,847 firmware images from 14 vendors, covering 8 device categories and 4 architectures. The corpus comprises 40 million procedures and 3,672 confirmed vulnerabilities, providing reliable ground truth for validation.

The experimental infrastructure included a 256-core cluster with 1TB of RAM. Despite the large dataset, processing 40 million procedures remained tractable. QEMU emulation supported dynamic testing across architectures. Docker containers ensured analysis isolation, preventing cross-contamination. Redis queues managed task distribution, and PostgreSQL stored results efficiently.

Dataset splitting followed standard practice: 70% training, 15% validation, and 15% testing. Stratification preserved the distribution of vulnerability types. For instance, buffer overflow vulnerabilities representing 10% of training data also comprised 10% of test data, preventing overfitting to specific vulnerability distributions.

5.2. Performance Metrics and Comparative Analysis

The experimental results demonstrate that our framework achieved a 93.7% true positive rate, missing only 6.3% of vulnerabilities. The false positive rate was 12.3%, down from a 28.4% baseline, representing a 56.7% relative reduction. Each percentage point corresponds to hundreds of false alarms eliminated, saving analyst hours. Weighted averages across device categories show high performance: consumer routers reached 95.2%, industrial controllers 94.1%, and medical devices 89.4%, the lower rate reflecting proprietary protocols and regulatory constraints.

Comparative baselines from lightweight static analysis achieved 76% detection with 19% false positives. The superior performance of our hybrid method stems from integrating dynamic validation, which effectively eliminates static false positives.

Average analysis time per firmware was 3.8 hours, 56.3% faster than exhaustive dynamic testing, with peak memory usage of 456MB during classification. These figures indicate that deployment is feasible on standard workstations rather than requiring specialized high-end clusters.

5.3. Case Studies on Real-World Firmware Samples

To evaluate practical effectiveness, we analyzed three representative domains.

Commercial Routers: Analysis revealed 47 zero-day vulnerabilities, including authentication bypasses in administrative interfaces, buffer overflows in DHCP handlers, and command injections in diagnostic tools. Each vulnerability was verified through proof-of-concept exploits, confirming the reliability of the detection framework.

Industrial Controllers: Critical flaws appeared in process control logic, including integer overflows in sensor processing and race conditions in alarm handlers. The priority ranking system successfully elevated safety-critical issues-those with potential physical consequences-above vulnerabilities affecting only system availability, demonstrating effective context-aware prioritization and alignment with operational risk [13].

Medical Devices: These devices posed distinct challenges due to proprietary protocols, stringent regulatory constraints, and non-standard architectures that initially complicated model predictions. Adaptations included custom protocol parsers, gentler fuzzing strategies, and architecture-specific feature extraction. Despite these complexities, the final detection accuracy reached 89.4%. Vulnerabilities impacting patient safety were consistently assigned maximum priority, validating the effectiveness of the context-aware ranking methodology [14,15].

6. Conclusion

This work presents a hybrid firmware vulnerability analysis framework that integrates static structural analysis with dynamic behavioral monitoring to achieve high detection accuracy while maintaining practical efficiency. By leveraging architecture-agnostic intermediate representations, probabilistic feature embeddings, and context-aware risk scoring, the framework addresses the challenges posed by stripped binaries, heterogeneous architectures, and resource-constrained embedded environments.

Extensive evaluation across 2,847 firmware images, encompassing 40 million procedures and multiple device categories-including consumer routers, industrial controllers, and medical devices-demonstrated a true positive rate of 93.7% and a false positive rate of 12.3%, representing a substantial improvement over baseline static or dynamic approaches. Case studies highlighted domain-specific performance variations, confirming the effectiveness of context-aware prioritization in elevating safety-critical vulnerabilities and guiding remediation efforts.

Furthermore, the proposed risk scoring methodology incorporates technical severity, operational context, dependency chains, and temporal factors to produce actionable vulnerability rankings, enabling efficient resource allocation and timely mitigation. Optimization techniques such as incremental analysis, sparse representations, and parallel processing ensure feasibility on memory- and computation-limited platforms.

In summary, this framework provides a practical, scalable, and robust solution for firmware security assessment, bridging the gap between theoretical detection capabilities and real-world operational requirements. Its combination of hybrid analysis and context-aware prioritization offers a blueprint for securing increasingly complex embedded systems, enhancing overall resilience against emerging vulnerabilities.

Acknowledgments: This investigation materialized through extensive collaboration. Industrial partners-who must remain anonymous for security reasons-provided thousands of firmware samples and validation expertise. Their trust enabled unprecedented analysis scope. The broader embedded security research community established foundations upon which we build. Anonymous reviewers challenged assumptions, strengthening arguments. Their critique transformed a good paper into-we hope-an excellent contribution. Funding arrived via National Science Foundation Grant CNS-2024789 and the Department of Defense Cyber Security Research Program. Computational resources flowed from our institution's High-Performance Computing Center-256 cores running continuously for three months, encompassing initial model training, hyperparameter optimization, cross-validation experiments, and the final evaluation of 2,847 firmware images. Open-source communities maintaining IDA Pro plugins, QEMU modifications, and binary analysis frameworks created the infrastructure making this research possible. To all contributors, named and unnamed, we extend sincere gratitude.

References

1. P. Sun, L. Garcia, G. Salles-Loustau, and S. Zonouz, "Hybrid firmware analysis for known mobile and IoT security vulnerabilities," In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June, 2020, pp. 373-384. doi: 10.1109/dsn48063.2020.00053
2. X. Feng, X. Zhu, Q. L. Han, W. Zhou, S. Wen, and Y. Xiang, "Detecting vulnerability on IoT device firmware: A survey," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 1, pp. 25-41, 2022.
3. M. Liu, Y. Zhang, J. Li, J. Shu, and D. Gu, "Security analysis of vendor customized code in firmware of embedded device," In *International Conference on Security and Privacy in Communication Systems*, October, 2016, pp. 722-739. doi: 10.1007/978-3-319-59608-2_40
4. Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 392-404, 2018.
5. A. Qasem, P. Shirani, M. Debbabi, L. Wang, B. Lebel, and B. L. Agba, "Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies," *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1-42, 2021.
6. J. B. Hou, T. Li, and C. Chang, "Research for vulnerability detection of embedded system firmware," *Procedia Computer Science*, vol. 107, pp. 814-818, 2017.
7. Y. Wang, J. Shen, J. Lin, and R. Lou, "Staged method of code similarity analysis for firmware vulnerability detection," *IEEE Access*, vol. 7, pp. 14171-14185, 2019. doi: 10.1109/access.2019.2893733
8. W. Xie, Y. Jiang, Y. Tang, N. Ding, and Y. Gao, "Vulnerability detection in IoT firmware: A survey," In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, December, 2017, pp. 769-772. doi: 10.1109/icpads.2017.00104

9. Y. G. Hassan, A. Collins, G. O. Babatunde, A. A. Alabi, and S. D. Mustapha, "Automated vulnerability detection and firmware hardening for industrial IoT devices," *International Journal of Multidisciplinary Research and Growth Evaluation*, vol. 4, no. 1, pp. 697-703, 2023. doi: 10.54660/.ijmrge.2023.4.1.697-703
10. S. Ul Haq, Y. Singh, A. Sharma, R. Gupta, and D. Gupta, "A survey on IoT & embedded device firmware security: Architecture, extraction techniques, and vulnerability analysis frameworks," *Discover Internet of Things*, vol. 3, no. 1, p. 17, 2023.
11. J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares," In *NDSS*, February, 2014, pp. 1-16. doi: 10.14722/ndss.2014.23229
12. T. Bakhshi, B. Ghita, and I. Kuzminykh, "A review of IoT firmware vulnerabilities and auditing techniques," *Sensors*, vol. 24, no. 2, p. 708, 2024. doi: 10.3390/s24020708
13. O. Sallenave, and R. Ducournau, "Lightweight generics in embedded systems through static analysis," *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 11-20, 2012. doi: 10.1145/2248418.2248421
14. H. Wang, Z. Ding, and Y. Zhong, "Static analysis test platform construction for embedded systems," In *2008 International Conference on Audio, Language and Image Processing*, July, 2008, pp. 808-812.
15. H. M. Kienle, J. Kraft, and T. Nolte, "System-specific static code analyses: A case study in the complex embedded systems domain," *Software Quality Journal*, vol. 20, no. 2, pp. 337-367, 2012. doi: 10.1007/s11219-011-9138-7