
**Arquitetura Reconfigurável (FPGA) Baseada
em Modelo de *Machine Learning* para Detecção
de Ransomware em Ambientes Pervasivos.**

Camilly de Melo Cruvinel



UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Camilly de Melo Cruvinel

**Arquitetura Reconfigurável (FPGA) Baseada
em Modelo de *Machine Learning* para Detecção
de Ransomware em Ambientes Pervasivos.**

Trabalho de Conclusão de Curso apresentado
à Faculdade de Computação da Universidade
Federal de Uberlândia, Minas Gerais, como
requisito exigido parcial à obtenção do Título
de Bacharel em Sistemas de Informação.

Área de concentração: Sistemas de Informação

Orientador: Diego Nunes Molinos

Uberlândia - MG

2026

*“O segredo da existência humana reside não só em viver, mas também em saber para
que se vive.”
(Fiódor Dostoiévski)*

Resumo

O crescente número de ataques de *ransomware* tem colocado em risco a segurança de dados e a disponibilidade de serviços em todo o mundo, com impacto financeiro estimado em bilhões de dólares anualmente. As abordagens tradicionais de detecção, baseadas em assinaturas, têm dificuldade em identificar novas variantes devido ao uso de ofuscação e polimorfismo, o que limita sua eficácia contra ataques de dia zero. Ao mesmo tempo, as arquiteturas de processadores de propósito geral carecem da flexibilidade de reconfiguração em nível de hardware necessária para o processamento adaptativo de modelos de Aprendizado de Máquina (AM) em ambientes de computação pervasiva, caracterizados por dispositivos com recursos de processamento, memória e energia limitados. Este trabalho propõe uma arquitetura de detecção de ransomware baseada em *Field-Programmable Gate Array* (FPGA), que combina técnicas de Aprendizado de Máquina com a implementação de um classificador em lógica combinacional, priorizando inferência determinística de baixa latência para dispositivos com recursos restritos. O método proposto compreende o treinamento e a comparação de cinco modelos supervisionados (Árvore de Decisão, Random Forest, *eXtreme Gradient Boosting* (XGBoost), Rede Neural e *Support Vector Machine* (SVM) sobre o *dataset* CIC-MalMem-2022, utilizando validação *hold-out* 80/20 combinada com validação cruzada de 5 *folds*, seguidos de análise de explicabilidade via *SHapley Additive exPlanations* (SHAP) para identificar o consenso entre os modelos quanto às características mais relevantes para a detecção. O modelo com melhor relação entre desempenho preditivo e simplicidade estrutural é então traduzido para uma descrição em *VHSIC Hardware Description Language* (VHDL), na qual os limiares de decisão aprendidos são incorporados como constantes de comparadores em ponto fixo, permitindo a síntese de um classificador de ciclo único em FPGA. Espera-se, com este trabalho, contribuir para o estado da arte em detecção de ransomware ao propor um modelo de hardware leve e de baixa latência, alternativo às arquiteturas de alto desempenho voltadas a ambientes de *data center* predominantes na literatura, e adequado à realidade de dispositivos embarcados e pervasivos com recursos computacionais restritos.

Palavras-chave: Ransomware, Aprendizado de Máquina, FPGA, Computação Pervasiva.

Abstract

The growing number of ransomware attacks has put data security and service availability at risk worldwide, causing an estimated financial impact of billions of dollars annually. Traditional signature-based detection approaches struggle to identify new variants due to obfuscation and polymorphism, limiting their effectiveness against zero-day attacks. At the same time, general-purpose processor architectures lack the hardware-level reconfiguration flexibility required for adaptive processing of Machine Learning (ML) models in pervasive computing environments, which are characterized by devices with limited processing, memory, and energy resources. This work proposes a ransomware detection architecture based on Field-Programmable Gate Array (FPGA) that combines Machine Learning techniques with the implementation of a classifier as combinational logic, prioritizing deterministic, low-latency inference for resource-constrained devices. The proposed method comprises training and comparing five supervised models (Decision Tree, Random Forest, eXtreme Gradient Boosting (XGBoost), Neural Network and Support Vector Machine (SVM)) on the CIC-MalMem-2022 dataset, using an 80/20 hold-out split combined with 5-fold cross-validation, followed by an explainability analysis using SHAP (SHapley Additive exPlanations) to identify the consensus among models regarding the most relevant features for detection. The model that best balances predictive performance and structural simplicity will then be translated into a VHSIC Hardware Description Language (VHDL) description, in which the learned decision thresholds are embedded as constants in fixed-point comparators, enabling the synthesis of a single-cycle FPGA classifier. This work is expected to contribute to the state of the art in ransomware detection by proposing a lightweight, low-latency hardware model suitable for embedded and pervasive devices with constrained computational resources, as an alternative to the high-performance, data-center-oriented architectures that predominate in the literature.

Keywords: Ransomware. Machine Learning. FPGA. Pervasive Computing.

Lista de ilustrações

Figura 1 – Visão geral do fluxo de trabalho metodológico	23
Figura 2 – Curva de aprendizado e matriz de confusão — Árvore de Decisão.	28
Figura 3 – Curva ROC — Árvore de Decisão.	29
Figura 4 – Árvore de decisão treinada, utilizada como base para a implementação em FPGA.	29
Figura 5 – Duas das árvores que compõem o <i>Random Forest</i> (limitadas à profundidade 4 para visualização).	30
Figura 6 – Curva de aprendizado e matriz de confusão — <i>Random Forest</i>	31
Figura 7 – Curva de aprendizado e matriz de confusão — XGBoost.	31
Figura 8 – Curva ROC — XGBoost.	32
Figura 9 – Curvas de perda e de acurácia por época — Rede Neural.	32
Figura 10 – Matriz de confusão — Rede Neural.	33
Figura 11 – Curva de aprendizado e matriz de confusão — <i>Support Vector Machine</i> (SVM).	34
Figura 12 – Curva ROC — SVM.	34
Figura 13 – Gráfico-resumo SHAP — Árvore de Decisão.	36
Figura 14 – Gráfico-resumo SHAP — <i>Random Forest</i>	36
Figura 15 – Gráfico-resumo SHAP — XGBoost.	37
Figura 16 – Gráfico-resumo SHAP — Rede Neural.	37
Figura 17 – Gráfico-resumo SHAP — SVM.	38
Figura 18 – Visão geral da arquitetura do classificador orientado a hardware em FPGA.	40

Lista de tabelas

Tabela 1 – Comparação entre abordagens de aceleração baseada em FPGA.	22
Tabela 2 – Performance dos Modelos de Aprendizado de Máquina.	34
Tabela 3 – Exemplo de tradução de um nó da árvore de decisão para lógica em FPGA.	39
Tabela 4 – Utilização de recursos de hardware no FPGA.	41
Tabela 5 – Métricas de Desempenho da Inferência de Software	42
Tabela 6 – Comparação de recursos: execução em software vs. FPGA.	44
Tabela 7 – Comparação do desempenho da inferência entre a implementação em software e FPGA.	44
Tabela 8 – Comparação da latência de inferência para diferentes cargas de trabalho.	45

Lista de siglas

AM Aprendizado de Máquina

FPGA *Field-Programmable Gate Array*

RNA Rede Neural Artificial

SHAP *SHapley Additive exPlanations*

SVM *Support Vector Machine*

VHDL VHSIC Hardware Description Language

XGBoost *eXtreme Gradient Boosting*

Sumário

1	INTRODUÇÃO	11
1.1	Justificativa	12
1.2	Hipótese de Pesquisa	13
1.3	Objetivos	13
1.4	Contribuições	14
1.5	Organização da Monografia	14
2	FUNDAMENTAÇÃO TEÓRICA E TRABALHOS CORRE-	
	LATOS	16
2.1	Computação Pervasiva	16
2.2	Ransomware	17
2.3	Aprendizado de Máquina, Algoritmos de Classificação e Expli-	
	cabilidade (SHAP)	18
2.3.1	Árvore de Decisão	18
2.3.2	Random Forest	18
2.3.3	XGBoost	19
2.3.4	Rede Neural Artificial	19
2.3.5	SVM	19
2.3.6	Explicabilidade e a Técnica SHAP	19
2.4	FPGAs e Aceleração de Hardware para Inferência de Modelos	
	de Aprendizado de Máquina	20
2.5	Trabalhos Correlatos	21
3	METODOLOGIA	23
3.1	Conjunto de Dados e Pré-processamento	23
3.2	Configuração dos Classificadores e Protocolo de Avaliação . . .	25
3.3	Projeto e Implementação em FPGA	26
3.4	Avaliação do Classificador em FPGA	27

4	TREINAMENTO DOS CLASSIFICADORES E EXPLICABILIDADE COM SHAP	28
4.1	Árvore de Decisão	28
4.2	Random Forest	30
4.3	XGBoost	31
4.4	Rede Neural Artificial	32
4.5	SVM	33
4.6	Desempenho Comparativo dos Classificadores	33
4.7	Análise de Explicabilidade (SHAP) e Seleção do Modelo	35
4.8	Síntese de Explicabilidade	38
5	PROJETO E IMPLEMENTAÇÃO DA ARQUITETURA EM FPGA	39
6	ANÁLISE DE INFERÊNCIA	42
6.1	Análise Temporal da Inferência	42
6.1.1	Impacto da Aritmética de Ponto Fixo no Desempenho de Classificação .	45
7	CONCLUSÃO	48
7.1	Trabalhos Futuros	49
	REFERÊNCIAS	51
	APÊNDICES	54
	APÊNDICE A – CÓDIGOS-FONTE DESENVOLVIDOS	55

Introdução

O crescente número de ataques de *ransomware* tem colocado em risco a segurança de dados em todo o mundo, destacando a necessidade urgente de soluções de detecção mais rápidas e eficientes. Ataques de *ransomware* afetaram aproximadamente uma em cada cinco empresas (21%), com o custo médio de reparação de um ataque chegando a US\$ 1,85 milhão (Hornetsecurity, 2022). Entre as ameaças emergentes, o *ransomware* se destaca por seu impacto operacional na disponibilidade de dados e serviços, afetando inclusive infraestruturas críticas (HUMAYUN et al., 2021). A eficácia desse tipo de ataque depende diretamente da velocidade de criptografia, tornando a detecção antecipada essencial para evitar danos irreversíveis (MALIK; KHANNA; SHARMA, 2024).

A detecção tradicional de *ransomware* depende, em grande parte, de técnicas baseadas em assinaturas, que identificam *malware* por meio de padrões conhecidos. Essas abordagens, no entanto, têm dificuldade em detectar novas variantes devido à ofuscação e ao polimorfismo, o que limita sua capacidade de identificar ataques de dia zero (ALRAIZZA; ALGARNI, 2023). Em resposta, abordagens baseadas em Aprendizado de Máquina (AM) têm sido amplamente exploradas para a detecção automatizada de ameaças; dentre elas, a análise de memória volátil destaca-se por capturar o comportamento do *malware* em tempo de execução, revelando artefatos que a análise estática não alcança, especialmente diante de amostras ofuscadas (CARRIER et al., 2022). Contudo, a maioria das soluções existentes depende de *frameworks* de alto nível e executa sobre sistemas operacionais convencionais, introduzindo sobrecarga computacional, maior latência de inferência e elevado consumo energético. Além disso, as arquiteturas de processadores de propósito geral carecem de flexibilidade de reconfiguração em nível de hardware, o que restringe sua eficácia no processamento adaptativo e otimizado de modelos de AM voltados à detecção dinâmica de *ransomware* (ALRAWASHDEH; PURDY, 2018). Essas limitações se agravam justamente nos ambientes pervasivos, nos quais a detecção em tempo real é crítica para a mitigação imediata dos ataques.

1.1 Justificativa

No cenário em que os recursos computacionais são escassos, os *Field-Programmable Gate Arrays* (FPGAs) emergem como uma alternativa promissora devido à sua capacidade de reconfiguração em nível de circuito, podem ser adaptados de forma ágil para responder tanto à natureza polimórfica do *ransomware* quanto à evolução constante dos algoritmos de AM, viabilizando a execução em hardware dedicado, com redução da latência de inferência e maior eficiência energética em comparação com soluções baseadas em software (NGO et al., 2021).

Abordagens que se apoiam em FPGAs permitem que os mecanismos de detecção sejam executados diretamente na malha reconfigurável, reduzindo a dependência de sistemas operacionais e de *frameworks* complexos. A literatura recente apresenta diversas abordagens de aceleração por FPGA capazes de alcançar alto *throughput* na inferência de modelos de AM baseados em árvores, como *eXtreme Gradient Boosting* (XGBoost) e *ensembles* (GAJJAR et al., 2024; ALCOLEA; RESANO, 2021; KULAGA; GORGON, 2014). Tais abordagens, entretanto, são voltadas majoritariamente a ambientes empresariais ou de *data center*, empregando *pipelines* de múltiplos estágios, múltiplas unidades de computação e hierarquias de memória complexas, o que exige recursos de hardware significativos, inviabilizando sua adoção em dispositivos pervasivos de baixo custo. Por outro lado, trabalhos que exploram a detecção precoce de *ransomware* por meio de contadores de desempenho de hardware, como o de Anand, Charan e Shukla (2023), obtêm bons resultados de acurácia, mas dependem inteiramente de execução em CPU, do sistema operacional e de *drivers* de coleta, não eliminando a dependência de software nem oferecendo latência determinística.

Destarte, entende-se como uma contribuição científica e tecnológica o projeto e implementação de uma arquitetura de detecção de *ransomware* que combine (i) a robustez preditiva de modelos de AM devidamente comparados e validados, (ii) uma camada de explicabilidade capaz de justificar a escolha do modelo e das características utilizadas, e (iii) um projeto de hardware leve, com inferência determinística de baixa latência, adequado a dispositivos com recursos restritos, características típicas de ambientes de computação pervasiva.

Nesse contexto, torna-se importante ressaltar que, a seleção do modelo a ser embarcado não deve se fundamentar exclusivamente em métricas globais de desempenho. Algoritmos distintos podem apresentar resultados numericamente equivalentes, embora construam suas predições a partir de mecanismos e características diferentes, nem sempre igualmente adequados à implementação em lógica de hardware. Técnicas de explicabilidade, especialmente a análise baseada em *SHapley Additive exPlanations* (SHAP) (LUNDBERG; LEE, 2017), permitem quantificar a contribuição individual de cada atributo para as predições e, assim, estabelecer uma base de comparação entre modelos de diferentes naturezas. Essa análise não apenas fundamenta a escolha do classificador, mas também orienta a redução

de dimensionalidade necessária à implementação embarcada e auxilia na identificação de correlações espúrias, constituindo uma prática recomendada na literatura de AM aplicada à segurança (ARP et al., 2022).

1.2 Hipótese de Pesquisa

A hipótese deste trabalho é que um classificador de AM com baixa complexidade estrutural, selecionado com base em critérios de desempenho preditivo e explicabilidade, pode ser implementado em FPGA como um mecanismo eficaz de detecção de *ransomware*, viabilizando inferência determinística de baixa latência, com implementação predominantemente em lógica combinacional observando o uso reduzido de recursos de hardware.

Para avaliar essa hipótese, diferentes algoritmos de AM são treinados e avaliados sob o mesmo protocolo experimental, utilizando o conjunto de dados CIC-MalMem-2022 (CARRIER et al., 2022). Os modelos são comparados com base em métricas de desempenho preditivo e interpretados por meio da análise de explicabilidade com SHAP, a fim de subsidiar a seleção do classificador mais adequado para implementação em FPGA.

A hipótese será corroborada caso um classificador de baixa complexidade estrutural seja selecionado e demonstre desempenho compatível com os requisitos da aplicação, aliado à viabilidade de uma implementação eficiente em hardware. Em contrapartida, a hipótese será refutada caso modelos de maior complexidade apresentem vantagens preditivas que justifiquem seu custo computacional adicional, ou caso o classificador selecionado não permita uma implementação em FPGA com os ganhos esperados em termos de simplicidade, determinismo e utilização de recursos.

Este trabalho é guiado pelas seguintes questões de pesquisa:

- a) Qual dos modelos avaliados apresenta a melhor relação entre desempenho preditivo e simplicidade estrutural para a detecção de *ransomware* a partir do *dataset* CIC-MalMem-2022?
- b) O que a análise de explicabilidade SHAP revela sobre os atributos que sustentam as predições de cada modelo?
- c) A implementação do classificador em FPGA proporciona tempo de inferência menor do que o de sua execução em software, sem perda significativa de acurácia?

1.3 Objetivos

O objetivo principal deste trabalho é implementar e avaliar uma arquitetura baseada em FPGA para a detecção de *ransomware* a partir de dados de memória, utilizando um classificador de aprendizado de máquina selecionado por meio da avaliação comparativa de diferentes algoritmos e da análise de explicabilidade com SHAP.

Como objetivos específicos, pretende-se:

1. realizar o pré-processamento e a seleção de *features* do *dataset* CIC-MalMem-2022;
2. treinar cinco modelos de AM utilizando validação *hold-out* 80/20 combinada com validação cruzada ($k\text{-fold} = 5$);
3. Aplicar a técnica SHAP para explicar as decisões de cada modelo e identificar as *features* mais influentes;
4. Implementar em FPGA a arquitetura do classificador;
5. Avaliar o tempo de inferência do modelo em nível de software e da arquitetura em FPGA;

1.4 Contribuições

Este trabalho pretende contribuir cientificamente com: (i) uma comparação sistemática entre cinco algoritmos de AM para detecção de *ransomware* sobre o *dataset* CIC-MalMem-2022, apoiada em análise de explicabilidade via SHAP; (ii) uma proposta de arquitetura de hardware leve, baseada em um classificador combinacional de ciclo único em FPGA, como alternativa às arquiteturas de alto desempenho e uso intensivo de recursos predominantes na literatura de aceleração por FPGA para modelos baseados em árvores (GAJJAR et al., 2024; ALCOLEA; RESANO, 2021; KULAGA; GORGON, 2014); e (iii) um modelo metodológico para a rápida tradução de modelos de AM previamente validados em soluções de hardware especializado, voltado especificamente a dispositivos com recursos restritos em ambientes de computação pervasiva.

É importante ressaltar ainda que este trabalho de graduação foi publicado no SB-CUP - 18º Simpósio Brasileiro de Computação Ubíqua e Pervasiva (SBCUP 2026) como artigo completo na seção técnica de Segurança Cibernética e Resiliência em Redes e Sistemas IoT. *Camilly Cruvinel, Radharani Rocha, Camilla Quincozes, and Diego Molinos. 2026. An FPGA-Accelerated Decision Tree Model for Ransomware Detection in Pervasive Environments. In Anais do XVIII Simpósio Brasileiro de Computação Ubíqua e Pervasiva, julho 19, 2026, Gramado/RS, Brasil. SBC, Porto Alegre, Brasil, 46-57. DOI: <https://doi.org/10.5753/sbcup.2026.21471>. (CRUVINEL et al., 2026)*

1.5 Organização da Monografia

Além deste capítulo introdutório, esta monografia está organizada da seguinte forma. O Capítulo 2 apresenta a fundamentação teórica necessária à compreensão do trabalho, abordando os conceitos de computação pervasiva, *ransomware*, algoritmos de AM e explicabilidade SHAP, e aceleração de hardware via FPGA e discute os principais trabalhos

relacionados, destacando suas contribuições e limitações frente à proposta deste trabalho, e apresenta uma tabela comparativa entre eles. O Capítulo 3 descreve o método de pesquisa adotado, detalhando o conjunto de dados utilizado, a configuração dos classificadores de AM, o protocolo de avaliação, o uso da análise SHAP e o projeto da arquitetura em FPGA. O Capítulo 4 reporta os experimentos e a análise dos resultados, comparando o desempenho dos modelos e fundamentando a seleção da Árvore de Decisão por meio da explicabilidade. O Capítulo 5 descreve o projeto e a implementação da arquitetura em FPGA, detalhando o mapeamento dos nós da árvore de decisão para lógica combinacional em VHSIC Hardware Description Language (VHDL) e a utilização de recursos de hardware. O Capítulo 6 apresenta a análise de inferência, comparando o desempenho temporal e a eficiência computacional da implementação em FPGA frente à execução equivalente em software. Por fim, o Capítulo 7 apresenta as conclusões, as principais contribuições efetivamente alcançadas e as perspectivas de trabalhos futuros.

Fundamentação Teórica e Trabalhos Correlatos

Este capítulo reúne os conceitos necessários à compreensão da proposta. A Seção 2.1 caracteriza a computação pervasiva e as restrições que ela impõe aos mecanismos de segurança. A Seção 2.2 descreve o *ransomware* como ameaça e as limitações da detecção baseada em assinaturas. A Seção 2.3 apresenta o AM, os cinco algoritmos de classificação avaliados neste trabalho e a técnica de explicabilidade SHAP. A Seção 2.4 discute o uso de FPGAs para aceleração da inferência de modelos de AM. Por fim, a Seção 2.5 analisa os trabalhos correlatos e posiciona a contribuição desta monografia.

2.1 Computação Pervasiva

A computação pervasiva, ou ubíqua, corresponde ao paradigma no qual a capacidade de processamento é incorporada de forma transparente ao ambiente físico, de modo que sensores, sistemas embarcados e dispositivos móveis operem de maneira contínua e integrada, sem exigir a atenção explícita do usuário (SATYANARAYANAN, 2002). Nessa visão, a computação deixa de estar concentrada em estações de trabalho convencionais e passa a se distribuir por uma malha heterogênea de dispositivos interconectados, muitos deles com severas restrições de processamento, memória e energia.

Essa onipresença computacional, entretanto, amplia consideravelmente a superfície de ataque (HAQUE; AHAMED, 2006). Ambientes pervasivos tipicamente mantêm sistemas de arquivos persistentes e expõem interfaces de rede, o que os torna suscetíveis a *malware* e, em particular, a ataques de *ransomware* (PARK et al., 2022; AZMOODEH et al., 2018). A segurança nesses cenários apresenta características próprias: os dispositivos frequentemente não dispõem de recursos suficientes para executar mecanismos de defesa tradicionais, baseados em *frameworks* de software de alto nível e em verificações contínuas sobre o sistema operacional. Como consequência, soluções de detecção voltadas a esse contexto precisam ser leves, deterministas e capazes de operar com baixa latên-

cia, requisitos que motivam a investigação de abordagens implementadas diretamente em hardware.

2.2 Ransomware

O *ransomware* é uma classe de *malware* cujo objetivo é restringir o acesso legítimo a dados ou serviços, normalmente por meio da criptografia de arquivos da vítima, exigindo o pagamento de um resgate para a sua liberação. Diferentemente de outras ameaças, seu impacto recai diretamente sobre a disponibilidade dos dados, podendo comprometer inclusive infraestruturas críticas (HUMAYUN et al., 2021). A eficácia do ataque está diretamente associada à velocidade de criptografia, o que torna a detecção antecipada essencial para evitar danos irreversíveis (MALIK; KHANNA; SHARMA, 2024).

Ao longo do tempo, o *ransomware* evoluiu de ataques direcionados a computadores pessoais para uma ameaça mais ampla, que atinge dispositivos móveis, ambientes de Internet das Coisas e sistemas ciberfísicos (OZ et al., 2022). Essa evolução é evidenciada por variantes específicas de plataforma, como o *ransomware* para *Android* e *Linux* (LIPOVSKÝ; STEFANKO; BRANISA, 2016; MOHURLE; PATIL, 2017), e por campanhas recentes baseadas no modelo de *Ransomware-as-a-Service* (RaaS), que revelam uma tendência a estratégias de ataque agnósticas de plataforma e orientadas a serviço (ALWASHALI; RAHMAN; ISMAIL, 2021). À medida que os ambientes pervasivos integram múltiplos componentes, o *ransomware* consolida-se como uma ameaça multiplataforma que afeta tanto ativos digitais quanto serviços operacionais.

A detecção tradicional de *ransomware* apoia-se majoritariamente em técnicas baseadas em assinaturas, que identificam o *malware* por meio de padrões conhecidos ou de trechos de código previamente catalogados (ALZAHIRANI et al., 2025). Embora eficazes contra ameaças já conhecidas, tais abordagens têm dificuldade em identificar novas variantes, dada a utilização de técnicas de ofuscação e polimorfismo, o que limita sua capacidade de detectar ataques de dia zero (ALRAIZZA; ALGARNI, 2023). Essa limitação motivou o emprego de abordagens comportamentais, capazes de reconhecer padrões de execução em vez de assinaturas estáticas. Nesse contexto, a análise de memória volátil destaca-se por capturar o comportamento do *malware* em tempo de execução, revelando artefatos que a análise estática não alcança, sobretudo diante de amostras ofuscadas (CARRIER et al., 2022).

2.3 Aprendizado de Máquina, Algoritmos de Classificação e Explicabilidade (SHAP)

O AM consiste no conjunto de técnicas que permitem a um sistema inferir padrões a partir de dados, sem que as regras de decisão sejam explicitamente programadas. Na detecção de *ransomware*, o AM surge como alternativa às assinaturas estáticas, pois aprende padrões de comportamento associados a amostras maliciosas e benignas (ALRAIZZA; ALGARNI, 2023). Contudo, muitas soluções baseadas em AM dependem de *frameworks* complexos e de recursos computacionais significativos, o que dificulta sua implantação em ambientes com restrição de recursos. A tarefa investigada nesta monografia é a de classificação binária: a partir de um vetor de características extraídas da memória, o modelo deve atribuir a cada amostra o rótulo *benigno* ou *ransomware*. A seguir, descrevem-se os cinco algoritmos avaliados neste trabalho.

2.3.1 Árvore de Decisão

A Árvore de Decisão é um classificador que organiza o processo de decisão em uma estrutura hierárquica de nós. Em cada nó interno, uma característica é comparada a um limiar, dividindo o espaço de atributos em regiões cada vez mais homogêneas até que se atinja um nó-folha, ao qual é associada uma classe (BREIMAN et al., 1984). O critério de divisão comumente empregado é a impureza de Gini, que quantifica a heterogeneidade das classes em um nó. Sua principal virtude, no contexto deste trabalho, é a baixa complexidade estrutural aliada à alta interpretabilidade: o caminho da raiz até a folha corresponde a uma sequência de comparações simples, diretamente traduzível para lógica combinacional em hardware. Essa característica distingue a árvore de decisão dos demais modelos e é central para a proposta de implementação em FPGA.

2.3.2 Random Forest

O *Random Forest* é um método de *ensemble* que combina múltiplas árvores de decisão treinadas sobre subconjuntos aleatórios dos dados e das características (*bagging* e seleção aleatória de atributos), agregando suas predições por votação majoritária (BREIMAN, 2001). Essa combinação reduz a variância e tende a melhorar a capacidade de generalização em relação a uma única árvore. Em contrapartida, a necessidade de avaliar dezenas ou centenas de árvores eleva o custo de inferência e a complexidade de um eventual mapeamento para hardware, uma vez que cada árvore demandaria sua própria estrutura de comparadores.

2.3.3 XGBoost

O XGBoost é uma implementação eficiente de *gradient boosting* sobre árvores, na qual os modelos são adicionados de forma sequencial, cada um corrigindo os erros residuais dos anteriores por meio da otimização de uma função objetivo regularizada (CHEN; GUESTRIN, 2016). O XGBoost costuma alcançar desempenho preditivo de ponta em dados tabulares, mas, assim como o *Random Forest*, baseia-se em um conjunto numeroso de árvores, o que implica maior custo computacional e maior consumo de recursos quando traduzido para uma arquitetura de hardware.

2.3.4 Rede Neural Artificial

As RNAs são modelos compostos por camadas de unidades de processamento (neurônios) interconectadas por pesos ajustáveis, capazes de aprender representações não lineares complexas por meio do algoritmo de retropropagação (LECUN; BENGIO; HINTON, 2015). Embora expressivas, essas redes envolvem grande número de multiplicações em ponto flutuante e demandam recursos consideráveis de memória e processamento, características pouco alinhadas às restrições de dispositivos pervasivos e ao objetivo de uma inferência determinista de ciclo único.

2.3.5 SVM

O SVM é um classificador que busca o hiperplano de separação de margem máxima entre as classes, podendo recorrer a funções de *kernel* para tratar problemas não linearmente separáveis (CORTES; VAPNIK, 1995). Trata-se de um modelo consolidado e teoricamente bem fundamentado, mas sua inferência envolve o cálculo de produtos internos com os vetores de suporte, o que, além de crescer com o número de vetores, dificulta a obtenção de uma implementação combinacional leve.

2.3.6 Explicabilidade e a Técnica SHAP

A seleção do modelo a ser embarcado não pode se basear exclusivamente em métricas globais de desempenho, pois algoritmos distintos podem atingir resultados numericamente equivalentes por caminhos internos muito diferentes, sendo que nem todos são igualmente viáveis quando traduzidos para lógica de hardware. Técnicas de explicabilidade permitem inspecionar quais atributos efetivamente sustentam as predições de cada modelo, oferecendo uma base comparável entre arquiteturas de naturezas distintas.

A técnica SHAP (LUNDBERG; LEE, 2017) fundamenta-se nos valores de *Shapley* da teoria dos jogos cooperativos (SHAPLEY, 1953) para atribuir a cada característica uma contribuição aditiva para a predição de uma amostra. Formalmente, a predição do modelo é decomposta como a soma de um valor-base com as contribuições individuais de cada

atributo, de modo que valores SHAP positivos deslocam a saída em direção a uma classe e valores negativos em direção à outra. Ao agregar essas contribuições sobre o conjunto de dados, obtém-se um *ranking* de importância global das características. No presente trabalho, o SHAP é empregado como uma verificação complementar (*double check*) sobre a comparação de desempenho: além de confirmar quais atributos dominam as decisões de cada modelo e de verificar a convergência dos *rankings* entre arquiteturas distintas, a análise evidencia que a árvore de decisão concentra suas predições em um número reduzido de características. Esse diagnóstico fundamenta a escolha do modelo destinado ao hardware e orienta a redução de dimensionalidade do classificador embarcado, prática recomendada na literatura de AM aplicada à segurança para o diagnóstico de correlações espúrias (ARP et al., 2022).

2.4 FPGAs e Aceleração de Hardware para Inferência de Modelos de Aprendizado de Máquina

Um FPGA é um dispositivo de lógica reconfigurável composto por blocos lógicos programáveis e interconexões que podem ser configurados após a fabricação para implementar circuitos digitais arbitrários. Diferentemente de processadores de propósito geral, que executam instruções de forma sequencial sobre uma arquitetura fixa, o FPGA permite descrever diretamente a estrutura do cálculo em hardware, explorando paralelismo de granularidade fina e oferecendo latência determinista.

Em cenários de detecção de *ransomware*, grandes volumes de dados comportamentais precisam ser analisados sob restrições rígidas de latência. Tais cargas de trabalho frequentemente envolvem padrões irregulares de acesso à memória e execução dependente de dados, o que reduz a eficiência em arquiteturas de propósito geral (GAJJAR et al., 2024). Nesse contexto, a aceleração baseada em FPGA apresenta-se como alternativa promissora: a reconfigurabilidade em nível de hardware e o paralelismo de granularidade fina permitem que a arquitetura seja adaptada às cargas de inferência de AM (MAKRANI et al., 2022; NGO et al., 2021), reduzindo a sobrecarga computacional, a latência e o consumo energético em comparação com soluções puramente baseadas em software (ALRAWASHDEH; PURDY, 2018). Essas características são particularmente adequadas a ambientes pervasivos, nos quais a execução eficiente e determinista é crítica.

Para modelos baseados em árvores, a tradução para hardware é especialmente direta: cada nó de decisão corresponde a uma comparação entre uma característica e um limiar constante, e o conjunto de comparações pode ser avaliado simultaneamente por lógica combinacional. O uso de aritmética de ponto fixo, em vez de ponto flutuante, elimina a necessidade de unidades aritméticas complexas e reduz ainda mais o consumo de recursos, viabilizando a inferência em ciclo único.

2.5 Trabalhos Correlatos

Diversos estudos exploram a aceleração de modelos de AM em FPGA, especialmente para modelos baseados em árvores. Gajjar et al. (2024) propõem o RD-FAXID, um acelerador de XGBoost em FPGA voltado à detecção de *ransomware*, capaz de alcançar alto *throughput* por meio de *pipelines* de múltiplos estágios. Alcolea e Resano (2021) apresentam um acelerador energeticamente eficiente para árvores de *gradient boosting*, empregando uma organização *multithread*. Kulaga e Gorgon (2014) exploram a avaliação paralela de *ensembles* de árvores em plataforma *Zynq*, com foco no aumento da vazão de inferência. Embora eficazes para grandes cargas de trabalho, tais arquiteturas são voltadas majoritariamente a ambientes empresariais ou de *data center*, empregando *pipelines* de múltiplos estágios, múltiplas unidades de computação e hierarquias de memória complexas, o que exige recursos de hardware significativos e inviabiliza sua adoção em dispositivos pervasivos de baixo custo.

Por outro lado, Anand, Charan e Shukla (2023) exploram a detecção precoce de *ransomware* por meio de contadores de desempenho de hardware, obtendo bons resultados de acurácia com um classificador *Random Forest*. Entretanto, a abordagem depende inteiramente de execução em CPU, do sistema operacional e de *drivers* de coleta, não eliminando a dependência de software nem oferecendo latência determinista.

A Tabela 1 sintetiza essas abordagens. Observa-se que os trabalhos existentes ou priorizam alto desempenho à custa de elevado consumo de recursos, ou mantêm a inferência atrelada à execução em software. Configura-se, assim, uma lacuna: a ausência de uma arquitetura de detecção de *ransomware* que combine robustez preditiva de modelos de AM devidamente comparados e validados, uma camada de explicabilidade capaz de justificar a escolha do modelo e das características utilizadas, e um projeto de hardware leve, com inferência determinista de baixa latência, adequado a dispositivos com recursos restritos. Esta monografia posiciona-se justamente nessa lacuna, ao materializar uma árvore de decisão como um classificador combinacional de ciclo único em FPGA, priorizando o projeto de hardware leve e a inferência determinista de baixa latência.

Tabela 1 – Comparação entre abordagens de aceleração baseada em FPGA.

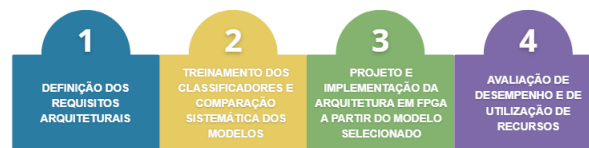
Trabalho	Modelo de AM	Plataforma	Projeto de HW	Contribuição principal
Gajjar et al. (2024)	XGBoost	FPGA	Pipeline de múltiplos estágios	Acelerador FPGA de alta vazão para inferência de árvores
Alcolea e Resano (2021)	GBDT	FPGA	Classe multithread	Acelerador FPGA energeticamente eficiente para <i>gradient boosting</i>
Kulaga e Gorgon (2014)	Ensembles	FPGA (Zynq)	Pipeline paralelo	Avaliação paralela de árvores para maior vazão
Anand, Charan e Shukla (2023)	Random Forest	CPU	Monitoramento HPC	Detecção precoce via contadores de desempenho de hardware
Este trabalho	Árvore de Decisão	FPGA	Classificador combinacional	Classificador FPGA de ciclo único para detecção de ransomware

Fonte: Elaborada pela autora (2026).

Metodologia

Esta pesquisa adota uma metodologia experimental orientada a projeto, na qual uma solução é concebida, implementada e avaliada para verificar seus ganhos em relação às abordagens existentes. A Figura 1 ilustra o fluxo de trabalho metodológico adotado neste trabalho.

Figura 1 – Visão geral do fluxo de trabalho metodológico



Fonte: Elaborada pela autora (2026).

3.1 Conjunto de Dados e Pré-processamento

Os experimentos utilizam o conjunto de dados CIC-MalMem-2022 (CARRIER et al., 2022; Canadian Institute for Cybersecurity, 2022), composto por características comportamentais extraídas da memória volátil a partir de execuções controladas de aplicações benignas e de amostras de malware ofuscado. Para sua construção, amostras de malware ofuscado, obtidas do repositório VirusTotal, e execuções de aplicações benignas, representativas do uso normal de um usuário, foram executadas separadamente em uma máquina virtual com o sistema operacional Microsoft Windows 10, sob o hipervisor VirtualBox. Durante cada execução, múltiplos despejos de memória da máquina virtual foram capturados externamente, em intervalos regulares, de modo a registrar o comportamento do processo ao longo do tempo de execução, e não apenas em um único instante.

Sobre cada despejo de memória foi aplicada a ferramenta *VolMemLyzer*, uma extensão do *framework* de análise forense *Volatility*, que percorre a imagem de memória e produz

um vetor de características numéricas organizado em grupos correspondentes aos módulos (*plugins*) de análise empregados: **pslist** (**pslist.***) (estatísticas sobre a lista de processos em execução, como número de processos e de *threads* por processo); **dlllist** (**dlllist.***) (quantidade e média de bibliotecas de vínculo dinâmico carregadas por processo); **handles** (**handles.***) (contagem de *handles* do sistema abertos por processo); **ldrmodules** (**ldrmodules.***) (indicadores de módulos carregados que não aparecem simultaneamente nas estruturas internas do Windows responsáveis por seu registro, uma técnica clássica de ocultação empregada por *malware*); **malfind** (**malfind.***) (indícios de injeção de código em regiões de memória executável não mapeadas a partir de um arquivo em disco); **psxview** (**psxview.***) (contagem de processos ausentes de determinadas listas internas do sistema operacional quando comparadas entre si, outro indicador de ocultação de processo); **svcsan** (**svcsan.***) (estatísticas sobre os serviços do Windows registrados, como quantidade de serviços em execução e de serviços associados a processos); e **callbacks** (**callbacks.***) (número de rotinas de retorno de chamada registradas no *kernel*, mecanismo frequentemente explorado por *rootkits* para interceptar eventos do sistema). Cada uma dessas categorias resume, portanto, um aspecto distinto do comportamento observável de um processo em memória, sem exigir sua execução estática nem a disponibilidade do binário original, o que é particularmente relevante para a detecção de *malware* ofuscado.

Cabe ressaltar uma limitação decorrente da origem do conjunto de dados: os despejos de memória que o compõem foram obtidos a partir de um sistema Windows 10 de propósito geral, cujas estruturas internas (listas de processos, tabelas de *handles*, módulos carregados, serviços do sistema, entre outras) diferem substancialmente das encontradas em sistemas operacionais tipicamente empregados em dispositivos embarcados e em ambientes de computação pervasiva, como distribuições Linux embarcado ou sistemas operacionais de tempo real (RTOS). Em tais plataformas, diversos artefatos capturados pelos módulos do *Volatility*, como a lista de serviços do Windows ou a tabela de módulos carregados no formato do kernel do Windows, não possuem correspondência direta, uma vez que esses sistemas adotam estruturas próprias de gerenciamento de processos, memória e módulos, em geral mais simples e com menor sobrecarga. Assim, o classificador treinado neste trabalho não deve ser interpretado como diretamente aplicável, sem adaptação, à análise de memória de dispositivos embarcados executando Linux ou RTOS. Seu emprego nesses ambientes exigiria a definição de um conjunto de características equivalente, extraído a partir das estruturas internas do sistema operacional-alvo, preservando a mesma lógica de análise comportamental da memória volátil. O uso do CIC-MalMem-2022 neste trabalho justifica-se, portanto, não pela correspondência direta ao ambiente de implantação final, mas pela disponibilidade de um conjunto de dados público, balanceado e amplamente utilizado na literatura de detecção de ransomware por análise de memória, adequado à etapa de seleção e comparação dos modelos de aprendizado de máquina que

precede a proposta da arquitetura de hardware. A arquitetura em FPGA proposta, por sua vez, é agnóstica ao conjunto específico de atributos utilizado, bastando que os comparadores de hardware sejam reconfigurados para um novo vetor de características, caso este trabalho seja estendido a um sistema operacional embarcado real.

Como o foco deste trabalho é a detecção de *ransomware*, foram utilizadas as amostras de *ransomware* do conjunto, incluindo as famílias Conti, MAZE, Pysa, Ako e Shade, em oposição a um número equivalente de amostras benignas, resultando em um conjunto balanceado de 19.580 instâncias. O balanceamento entre as classes foi adotado para evitar o viés do classificador em direção à classe majoritária e dispensa técnicas de reamostragem.

O pré-processamento consistiu na remoção de colunas não informativas para a classificação binária (identificadores textuais de categoria), na codificação do rótulo-alvo em formato binário (*benigno* = 0, *ransomware* = 1) e na verificação de valores ausentes ou constantes. Para os modelos sensíveis à escala das características (Rede Neural Artificial (RNA) e SVM), aplicou-se a padronização dos atributos; os modelos baseados em árvore (Árvore de Decisão, Random Forest e XGBoost) são invariantes a transformações monotônicas das características e, portanto, dispensam essa etapa.

3.2 Configuração dos Classificadores e Protocolo de Avaliação

Foram treinados e comparados cinco algoritmos de classificação: Árvore de Decisão, *Random Forest*, XGBoost, RNA e SVM, descritos na Seção 2.3. Cada modelo foi treinado com os parâmetros padrão fornecidos pelas respectivas bibliotecas de aprendizado de máquina, a fim de garantir uma comparação sob condições equivalentes e evitar que os resultados fossem enviesados por um ajuste fino específico de algum algoritmo. A Árvore de Decisão, a *Random Forest* e a SVM foram implementadas com a biblioteca *scikit-learn*; o XGBoost, com a biblioteca homônima *xgboost*, utilizando sua própria implementação do algoritmo de *gradient boosting*; e a RNA, com o framework *TensorFlow* e a API *Keras*, como um perceptron multicamadas treinado com a função de perda de entropia cruzada binária.

Para assegurar consistência e reprodutibilidade, adotou-se um protocolo experimental unificado, aplicado igualmente a todos os modelos. Utilizou-se validação *hold-out* na proporção 80/20 sobre as 19.580 instâncias, resultando em 15.664 amostras de treino e 3.916 amostras de teste. Sobre o conjunto de treino, empregou-se ainda validação cruzada estratificada com $k = 5$ (*5-fold*), o que permitiu estimar a estabilidade do desempenho e minimizar o risco de sobreajuste. O mesmo particionamento dos dados foi mantido entre os modelos, garantindo que as diferenças observadas decorressem exclusivamente dos algoritmos.

O desempenho foi caracterizado por um conjunto padronizado de métricas: acurácia, precisão, *recall*, *F1-score* e AUC-ROC. Complementarmente, foram geradas curvas de aprendizado e de perda, para diagnosticar sobreajuste (*overfitting*) e subajuste (*underfitting*), e matrizes de confusão, para inspecionar a distribuição de acertos e erros por classe.

Sobre os modelos treinados, aplicou-se a análise SHAP como verificação complementar (*double check*) à comparação de desempenho. Para os modelos baseados em árvore, empregou-se o explicador específico para estruturas de árvore; para a Rede Neural e a SVM, utilizou-se um explicador independente de modelo sobre uma amostra representativa do conjunto de teste. O objetivo dessa etapa foi triplo: (i) identificar as características que efetivamente sustentam as previsões de cada modelo; (ii) verificar a convergência dos *rankings* de importância entre arquiteturas de naturezas distintas; e (iii) avaliar quantas características são de fato necessárias a cada modelo, informação decisiva para a seleção do classificador a ser embarcado, já que o número de atributos relevantes determinou diretamente a quantidade de comparadores da arquitetura em hardware.

3.3 Projeto e Implementação em FPGA

A partir da comparação de desempenho e da análise SHAP, selecionou-se o classificador a ser materializado em hardware. A arquitetura proposta seguiu um modelo orientado a hardware no qual as características de entrada são avaliadas por uma sequência hierárquica de comparações que reproduz a estrutura da árvore de decisão treinada.

Cada nó de decisão da árvore foi mapeado para uma estrutura de comparação em ponto fixo descrita em VHDL. As características foram representadas em aritmética de ponto fixo com sinal de 32 bits, o que possibilitou comparações numéricas eficientes sem a sobrecarga de operações em ponto flutuante. Os valores de limiar extraídos do modelo treinado foram embutidos como constantes na descrição de hardware, permitindo que o classificador operasse essencialmente como lógica combinacional de comparação, seguida de um registrador de saída sincronizado ao *clock*. Como exemplo, a regra “se *característica* $\leq$ limiar então benigno, senão ransomware” foi convertida em uma operação de comparação digital sobre constantes de ponto fixo, sintetizada como comparadores baseados em LUTs no tecido do FPGA.

As saídas dos comparadores foram combinadas de acordo com a estrutura da árvore para gerar o sinal final de classificação. Como as comparações foram avaliadas em paralelo pela lógica combinacional, a arquitetura realizou a inferência em um único ciclo de *clock*, com complexidade de execução $O(1)$: a latência independe do tamanho do conjunto de dados ou da distribuição das entradas. Embora combinacional, a inferência permanece sujeita aos requisitos de temporização síncrona; o período de *clock* deve satisfazer $T_{clk} \geq T_{comb} + T_{setup} + T_{skew} + T_{jitter}$, em que T_{comb} representa o atraso do caminho crítico ao

longo dos comparadores, T_{setup} o tempo de estabilização do registrador e T_{skew} e T_{jitter} as incertezas de distribuição do *clock*. Dessa forma, a frequência de operação determina a vazão máxima do classificador.

3.4 Avaliação do Classificador em FPGA

A avaliação experimental compara o desempenho de inferência do classificador entre as implementações em software e em FPGA. Os experimentos em software foram conduzidos em uma estação Fedora Linux 41 equipada com um processador Intel Core i5-7200U (2,50 GHz), utilizando Python. A implementação em hardware teve como alvo um FPGA Intel Cyclone IV (EP4CE115F29C7), embarcado na placa Altera DE2-115 e operando a 50 MHz. A síntese foi realizada no Quartus Prime Lite Edition 22.1 e a verificação funcional no ModelSim.

Para garantir uma comparação, ambas as implementações processaram exatamente os mesmos subconjuntos do conjunto de teste, compostos pelos primeiros n registros, com tamanhos de lote iguais a 1, 50, 100, 150, 500 e 1000 instâncias. Esses lotes foram utilizados tanto na avaliação da inferência em software quanto na validação da implementação em FPGA. Para cada configuração, foram analisadas métricas de desempenho, incluindo latência média de inferência, tempo por instância, vazão (*throughput*), utilização de CPU e consumo de memória na implementação em software, além da latência de inferência, utilização de recursos lógicos e consumo estimado de potência na implementação em FPGA.

Antes das medições em software, foi executada uma etapa de aquecimento (*warm-up*), cujos resultados foram descartados para reduzir os efeitos de inicialização e estabilizar o ambiente de execução. O consumo de potência da implementação em FPGA foi estimado por meio do Power Analyzer do Quartus Prime após o processo de *place-and-route*. A comparação entre as implementações tem como objetivo avaliar os ganhos proporcionados pela arquitetura dedicada em termos de tempo de inferência, eficiência computacional e utilização de recursos de hardware.

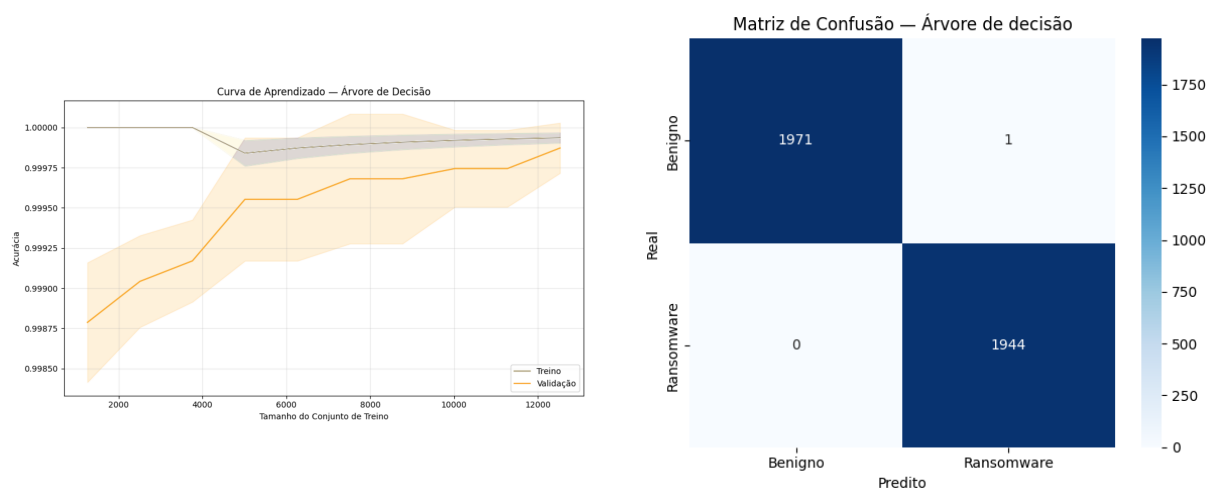
Treinamento dos Classificadores e Explicabilidade com SHAP

Este capítulo apresenta o treinamento dos classificadores, a explicabilidade via SHAP e comparação do desempenho preditivo dos cinco classificadores.

4.1 Árvore de Decisão

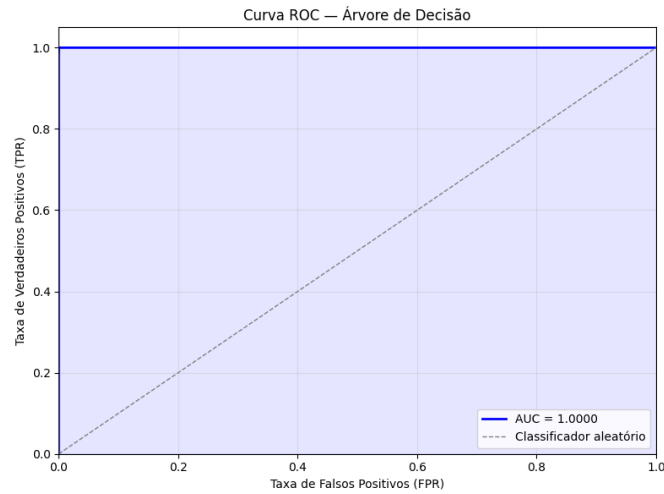
A Figura 2 apresenta a curva de aprendizado e a matriz de confusão da Árvore de Decisão. A curva mostra as acurácias de treino e de validação convergindo para valores muito próximos à medida que o conjunto de treino cresce, com margem reduzida entre elas, o que indica ausência de sobreajuste relevante. Na matriz de confusão, observa-se um único falso positivo e nenhum falso negativo, consistente com o *recall* de 1,0000. A curva ROC correspondente (Figura 3) atinge AUC de aproximadamente 1,0000.

Figura 2 – Curva de aprendizado e matriz de confusão — Árvore de Decisão.



Fonte: Elaborada pela autora (2026).

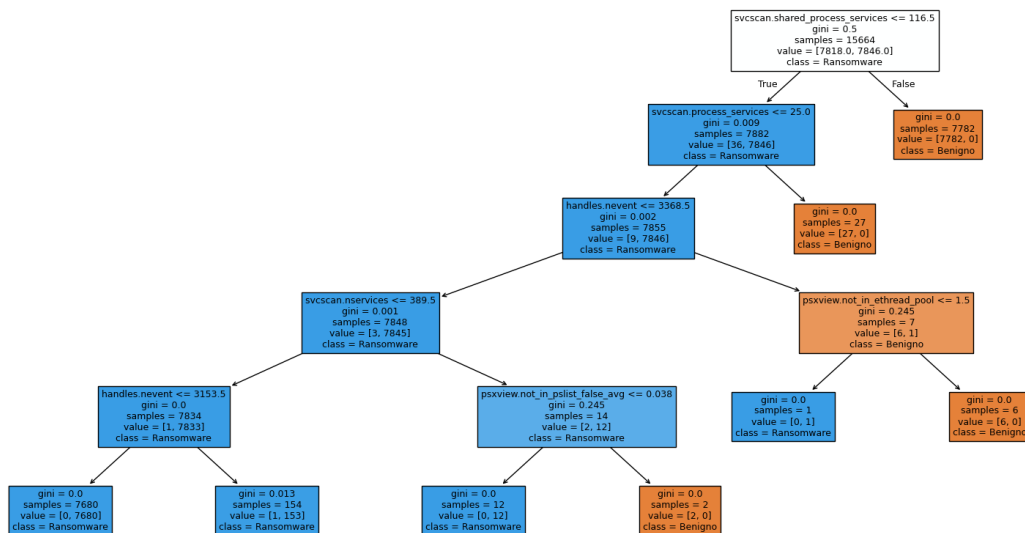
Figura 3 – Curva ROC — Árvore de Decisão.



Fonte: Elaborada pela autora (2026).

A Figura 4 exibe a árvore de decisão treinada. Nota-se que o modelo resolve a classificação a partir de um pequeno número de comparações sobre atributos relacionados a serviços do sistema e a *handles*, com nós-folha de alta pureza (impureza de Gini próxima de zero). Essa estrutura enxuta é precisamente o que viabiliza o mapeamento direto para lógica combinacional descrito no Capítulo 3.

Figura 4 – Árvore de decisão treinada, utilizada como base para a implementação em FPGA.

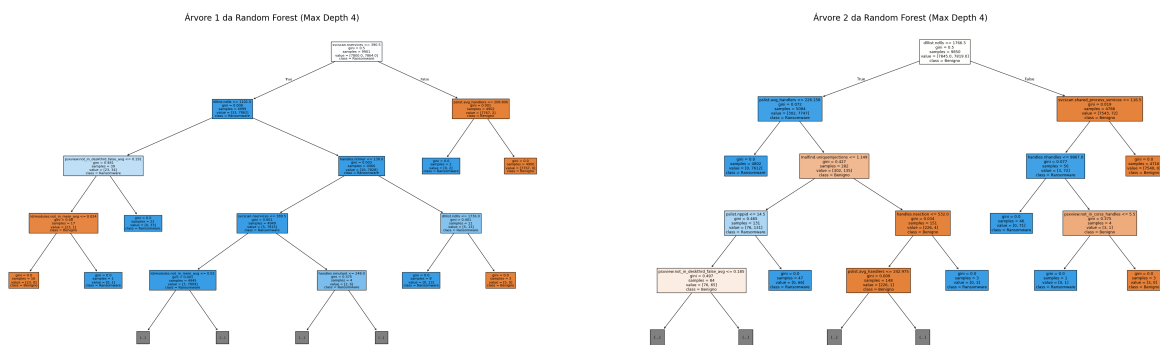


Fonte: Elaborada pela autora (2026).

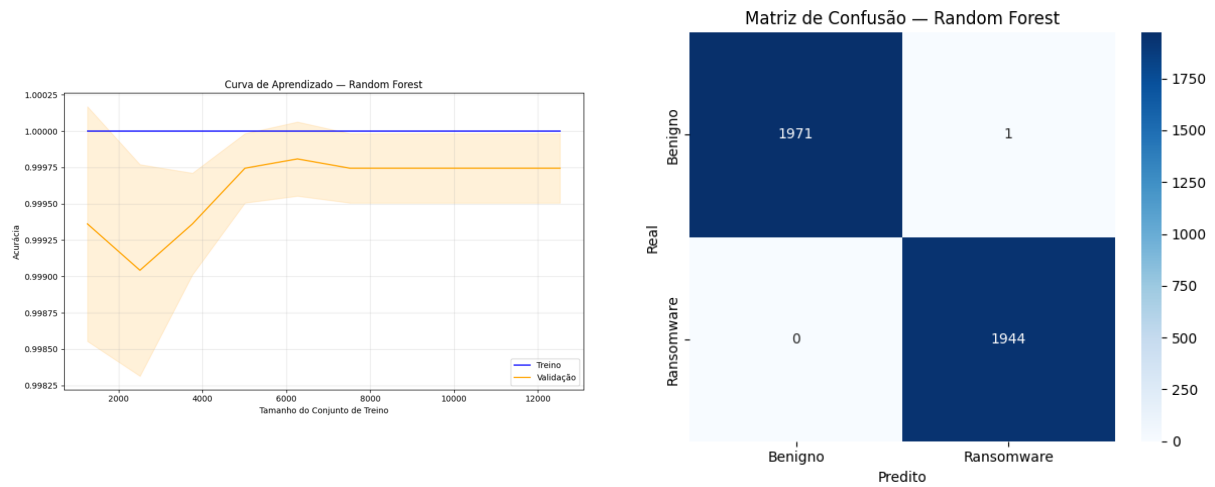
4.2 Random Forest

O *Random Forest* reproduz as métricas de teste da Árvore de Decisão (Tabela 2), com a mesma matriz de confusão, contendo um falso positivo e nenhum falso negativo (Figura 6). Sua curva de aprendizado mostra a acurácia de treino constante em 1,0000 e a de validação estabilizando pouco abaixo, comportamento típico de *ensembles* que memorizam o treino, mas mantêm boa generalização. A vantagem preditiva sobre a árvore única é, portanto, nula neste problema, ao passo que o custo de inferência e a complexidade de mapeamento para hardware seriam consideravelmente maiores, dada a necessidade de replicar a estrutura de comparadores para cada árvore do conjunto. Para ilustrar essa complexidade estrutural, a Figura 5 apresenta duas das árvores que compõem o *ensemble* (limitadas à profundidade 4 apenas para fins de visualização); cada uma delas já é substancialmente mais profunda e ramificada do que a árvore de decisão única, o que evidencia o custo de traduzir o modelo completo para hardware.

Figura 5 – Duas das árvores que compõem o *Random Forest* (limitadas à profundidade 4 para visualização).



Fonte: Elaborada pela autora (2026).

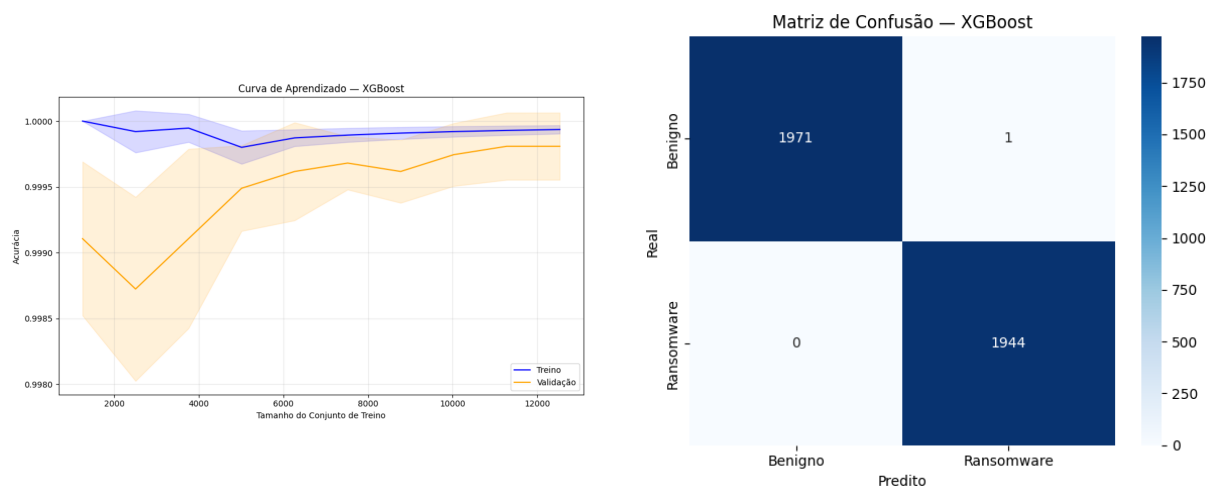
Figura 6 – Curva de aprendizado e matriz de confusão — *Random Forest*.

Fonte: Elaborada pela autora (2026).

4.3 XGBoost

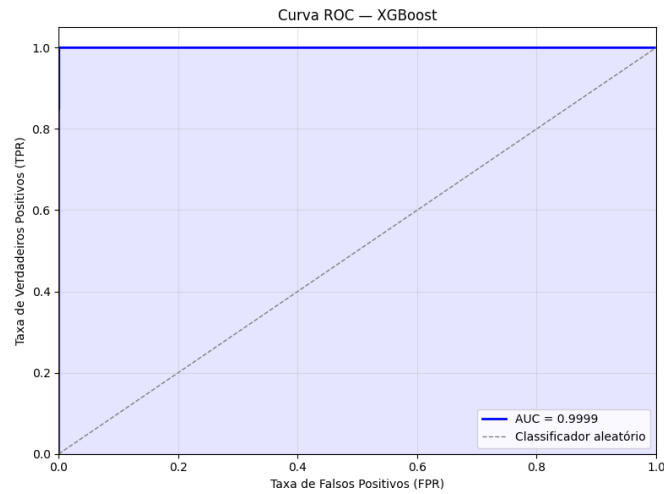
O XGBoost também iguala o desempenho dos demais modelos baseados em árvore, com acurácia de 0,9997 e AUC-ROC de 0,9999 (Figura 7 e Figura 8). Como no caso do *Random Forest*, o ganho preditivo em relação à árvore única é inexistente neste conjunto de dados, enquanto o custo de uma eventual implementação em hardware seria significativamente maior, por envolver a soma de múltiplas árvores de *boosting*.

Figura 7 – Curva de aprendizado e matriz de confusão — XGBoost.



Fonte: Elaborada pela autora (2026).

Figura 8 – Curva ROC — XGBoost.

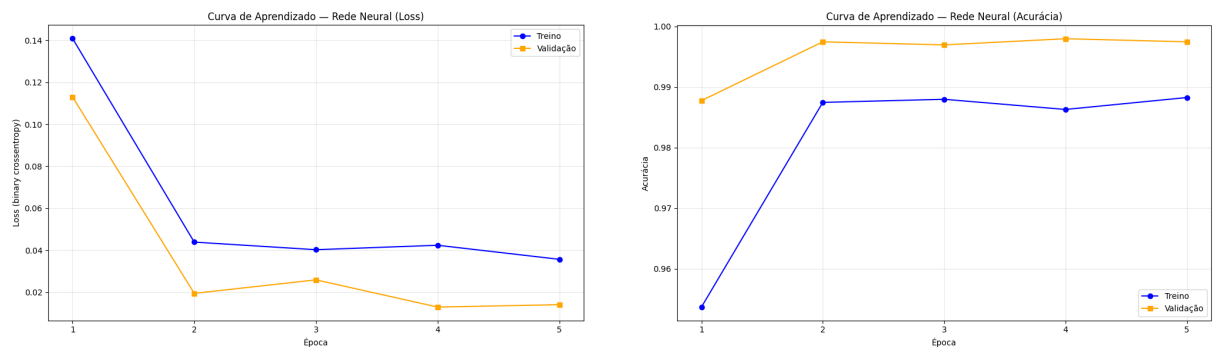


Fonte: Elaborada pela autora (2026).

4.4 Rede Neural Artificial

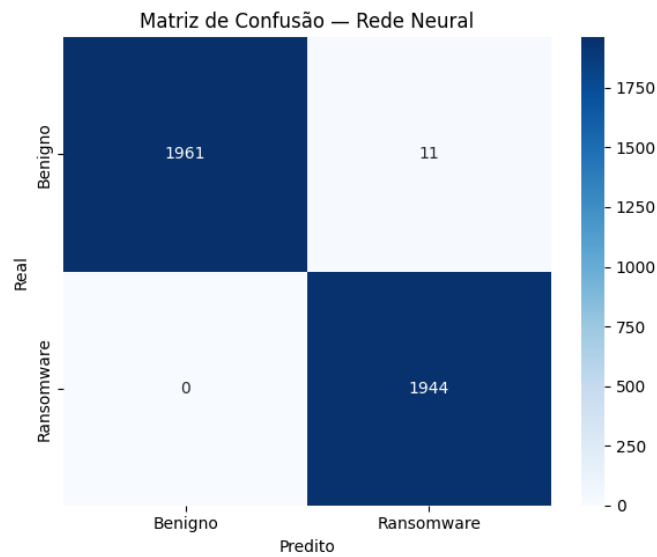
A Rede Neural apresenta acurácia de 0,9971, ligeiramente inferior à dos modelos baseados em árvore. Suas curvas de perda e de acurácia por época (Figura 9) mostram convergência estável, com a acurácia de validação acompanhando a de treino, sem sinais de sobreajuste. A matriz de confusão (Figura 10) registra 11 falsos positivos, mais do que os modelos baseados em árvore, ainda que sem falsos negativos. Apesar do bom desempenho, a rede envolve grande número de operações em ponto flutuante, o que a torna pouco aderente ao objetivo de uma inferência determinista de ciclo único em hardware leve.

Figura 9 – Curvas de perda e de acurácia por época — Rede Neural.



Fonte: Elaborada pela autora (2026).

Figura 10 – Matriz de confusão — Rede Neural.



Fonte: Elaborada pela autora (2026).

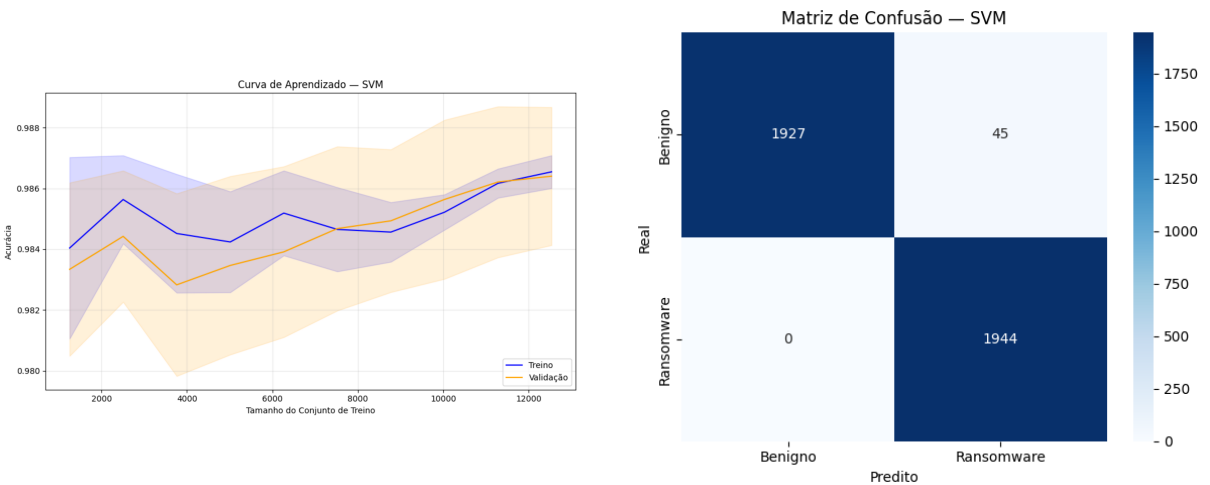
4.5 SVM

O SVM registrou o menor desempenho entre os modelos avaliados, com acurácia de 0,9885, F1-Score de 0,9886 e AUC-ROC de 0,9901 (Tabela 2). Sua curva de aprendizado (Figura 11) e a curva ROC (Figura 12) refletem esse patamar inferior: a separação entre as classes é boa, porém menos nítida do que a alcançada pelos modelos baseados em árvore, cuja AUC-ROC é praticamente unitária. A matriz de confusão (Figura 11) mantém o *recall* de 1,0000, mas apresenta um número de falsos positivos consideravelmente maior do que o dos demais modelos (cerca de 45, coerente com a precisão de 0,9774), o que indica uma tendência a classificar amostras benignas como *ransomware*. Além do desempenho inferior, a inferência do SVM depende do cálculo de produtos internos com os vetores de suporte, o que a torna pouco aderente ao objetivo de uma implementação combinacional leve e determinista em hardware.

4.6 Desempenho Comparativo dos Classificadores

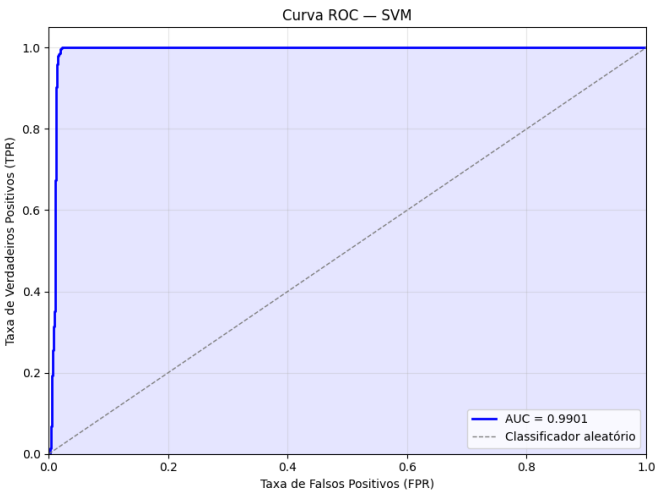
A Tabela 2 consolida as métricas de desempenho dos cinco modelos sobre o conjunto de teste (3.916 amostras), acompanhadas da média e do desvio-padrão da acurácia obtidos na validação cruzada estratificada (*5-fold*) sobre o conjunto de treino. Todos os modelos alcançaram *recall* de 1,0000, ou seja, nenhuma amostra de *ransomware* deixou de ser detectada, comportamento desejável em um cenário de segurança, no qual falsos negativos são especialmente custosos.

Figura 11 – Curva de aprendizado e matriz de confusão — SVM.



Fonte: Elaborada pela autora (2026).

Figura 12 – Curva ROC — SVM.



Fonte: Elaborada pela autora (2026).

Tabela 2 – Performance dos Modelos de Aprendizado de Máquina.

Modelo	Acurácia da Validação Cruzada						Métricas de Teste			
	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	Média	Precisão	F1-score	Acurácia	Recall
Árvore de Decisão	0,9997	1,0000	1,0000	1,0000	0,9997	0,9999	0,9994	0,9997	0,9997	1,0000
Random Forest	0,9994	1,0000	0,9997	1,0000	0,9997	0,9997	0,9994	0,9997	0,9997	1,0000
XGBoost	0,9997	1,0000	1,0000	1,0000	0,9994	0,9998	0,9994	0,9997	0,9997	1,0000
Rede Neural Artificial	0,9959	0,9962	0,9981	0,9968	0,9974	0,9969	0,9943	0,9971	0,9971	1,0000
SVM	0,9879	0,9856	0,9898	0,9831	0,9856	0,9864	0,9773	0,9885	0,9885	1,0000

Fonte: Elaborado pela autora (2026)

Os resultados evidenciam três patamares de desempenho. Os modelos baseados em árvore (Árvore de Decisão, *Random Forest* e XGBoost) apresentam desempenho pratica-

mente idêntico e superior, com acurácia de 0,9997 e AUC-ROC próxima da unidade. A Rede Neural situa-se logo abaixo, com acurácia de 0,9971, e o SVM apresenta o menor desempenho do conjunto, com acurácia de 0,9885 e AUC-ROC de 0,9901. Note-se que a Árvore de Decisão iguala numericamente os dois *ensembles* mais complexos (*Random Forest* e *XGBoost*), tanto nas métricas de teste quanto na estabilidade da validação cruzada, apesar de sua complexidade estrutural substancialmente menor. Esse é o primeiro indício a favor da seleção da árvore: ela oferece o melhor desempenho observado sem o custo dos modelos de conjunto.

4.7 Análise de Explicabilidade (SHAP) e Seleção do Modelo

A comparação da Seção 4.6 mostra que os três modelos baseados em árvore são numericamente equivalentes. Como discutido no Capítulo 3, esse empate de desempenho não é suficiente para decidir qual modelo embarcar, pois modelos distintos podem atingir os mesmos números por caminhos internos muito diferentes. A análise SHAP atua, portanto, como uma verificação complementar (*double check*), examinando *quais* e *quantas* características efetivamente sustentam as predições de cada modelo.

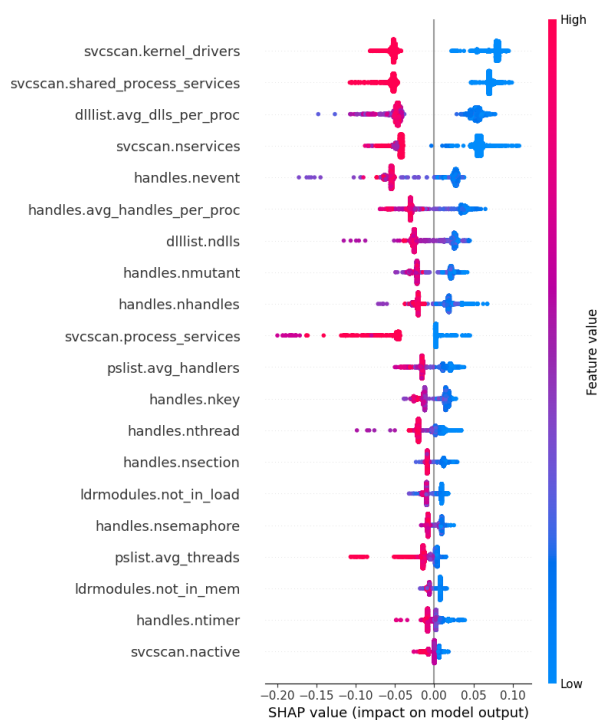
A Figura 13 apresenta o gráfico-resumo SHAP da Árvore de Decisão. Observa-se que a contribuição para as predições concentra-se de forma acentuada em um pequeno conjunto de atributos, encabeçado por `svcscan.shared_process_services`, seguido por `handles.nevent`, `svcscan.process_services` e `svcscan.nservices`; os demais atributos apresentam contribuição praticamente nula. Esse padrão é coerente com a estrutura da árvore treinada (Figura 4), que resolve a classificação com poucos nós de decisão. Em outras palavras, a Árvore de Decisão é o modelo que *menos* características necessita para operar.

Em contraste, os gráficos SHAP do *Random Forest* (Figura 14), do *XGBoost* (Figura 15), da Rede Neural (Figura 16) e do SVM (Figura 17) distribuem a importância por um número consideravelmente maior de atributos. Nos *ensembles*, características como `svcscan.kernel_drivers`, `dlllist.avg_dlls_per_proc`, `handles.avg_handles_per_proc` e diversos *handles* passam a contribuir de maneira relevante, refletindo o fato de que esses modelos combinam muitas árvores e, conseqüentemente, exploram um espectro mais amplo de atributos. A RNA e o SVM exibem comportamento análogo, com a importância diluída entre vários atributos.

Figura 13 – Gráfico-resumo SHAP — Árvore de Decisão.

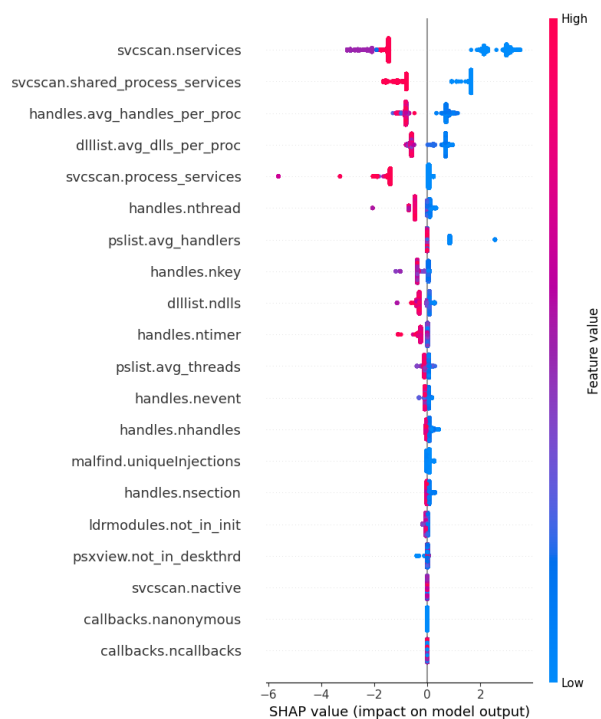


Fonte: Elaborada pela autora (2026).

Figura 14 – Gráfico-resumo SHAP — *Random Forest*.

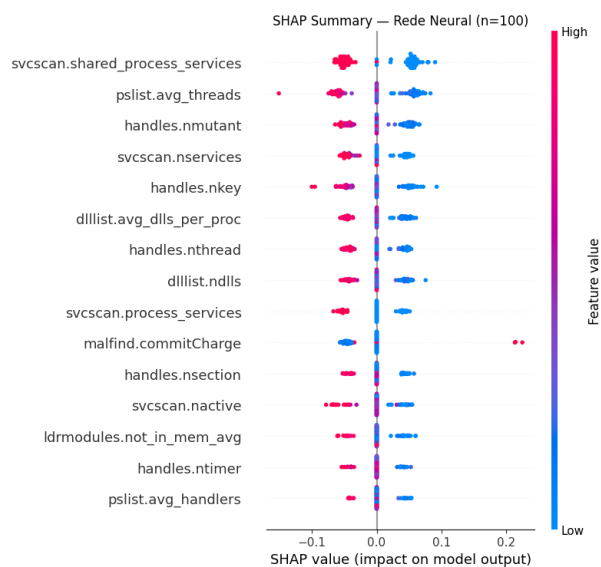
Fonte: Elaborada pela autora (2026).

Figura 15 – Gráfico-resumo SHAP — XGBoost.



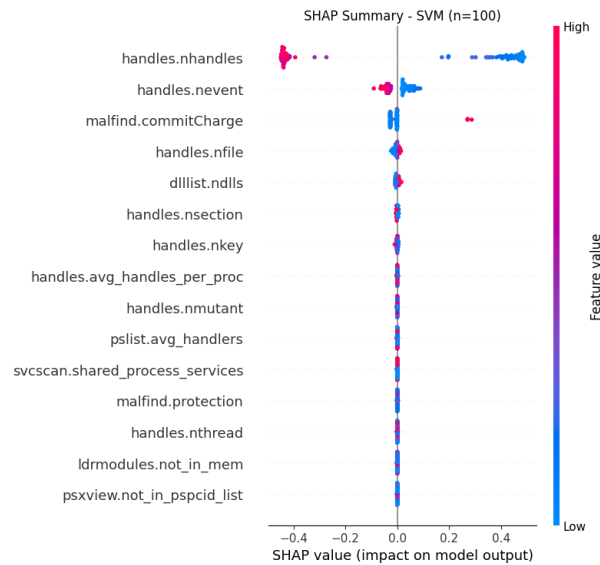
Fonte: Elaborada pela autora (2026).

Figura 16 – Gráfico-resumo SHAP — Rede Neural.



Fonte: Elaborada pela autora (2026).

Figura 17 – Gráfico-resumo SHAP — SVM.



Fonte: Elaborada pela autora (2026).

4.8 Síntese de Explicabilidade

Dois achados sustentam a decisão de projeto. Primeiro, há convergência dos atributos dominantes entre arquiteturas de naturezas distintas: características associadas a serviços do sistema (em especial `svcscan.shared_process_services`, `svcscan.process_services` e `svcscan.nservices`) figuram entre as mais importantes em todos os modelos. Essa convergência funciona como o *double check*, indicando que tais atributos são genuinamente discriminativos para a separação entre *ransomware* e software benigno, e não fruto de correlações espúrias captadas por um único modelo. Segundo, entre os modelos que atingem o melhor desempenho, a Árvore de Decisão é a que depende do menor número de características, concentrando quase toda a sua capacidade preditiva nesse pequeno núcleo consensual de atributos.

A conjunção desses dois fatores fundamenta a seleção da Árvore de Decisão como o modelo a ser embarcado. Ela reúne o melhor desempenho preditivo observado, empatando com os *ensembles* mais complexos, a menor complexidade estrutural e a menor dependência de atributos, propriedade decisiva para a implementação em hardware, uma vez que cada característica relevante corresponde a um comparador na arquitetura combinacional. Assim, a árvore permite a inferência com o menor número de comparadores e, portanto, com a menor utilização de recursos, sem sacrifício de acurácia.

Projeto e Implementação da Arquitetura em FPGA

Entre os modelos treinados , foi selecionado aquele que apresentou a melhor relação entre capacidade de classificação e custo de implementação em hardware. Essas características contribuem para uma implementação mais eficiente em FPGA, reduzindo o número de comparadores, a lógica combinacional necessária e o consumo de recursos, sem comprometer o desempenho obtido na etapa de treinamento.

Foi considerado para a construção da arquitetura, o modelo de árvore de decisão. Cada nó interno da árvore foi mapeado para um comparador em ponto fixo, responsável por avaliar um atributo em relação ao respectivo limiar (*threshold*), enquanto cada folha representa a decisão final de classificação, indicando se a amostra é classificada como *ransomware* ou *benigna*.

Dessa forma, o caminho percorrido pelos comparadores reproduz fielmente a avaliação hierárquica realizada pelo modelo treinado, permitindo que a inferência seja executada diretamente no hardware. A Tabela 3 ilustra esse processo de tradução, tomando um nó como exemplo: a regra da árvore torna-se uma operação de comparação, o limiar aprendido é embutido como constante em ponto fixo e a saída corresponde a um sinal binário de classificação.

Tabela 3 – Exemplo de tradução de um nó da árvore de decisão para lógica em FPGA.

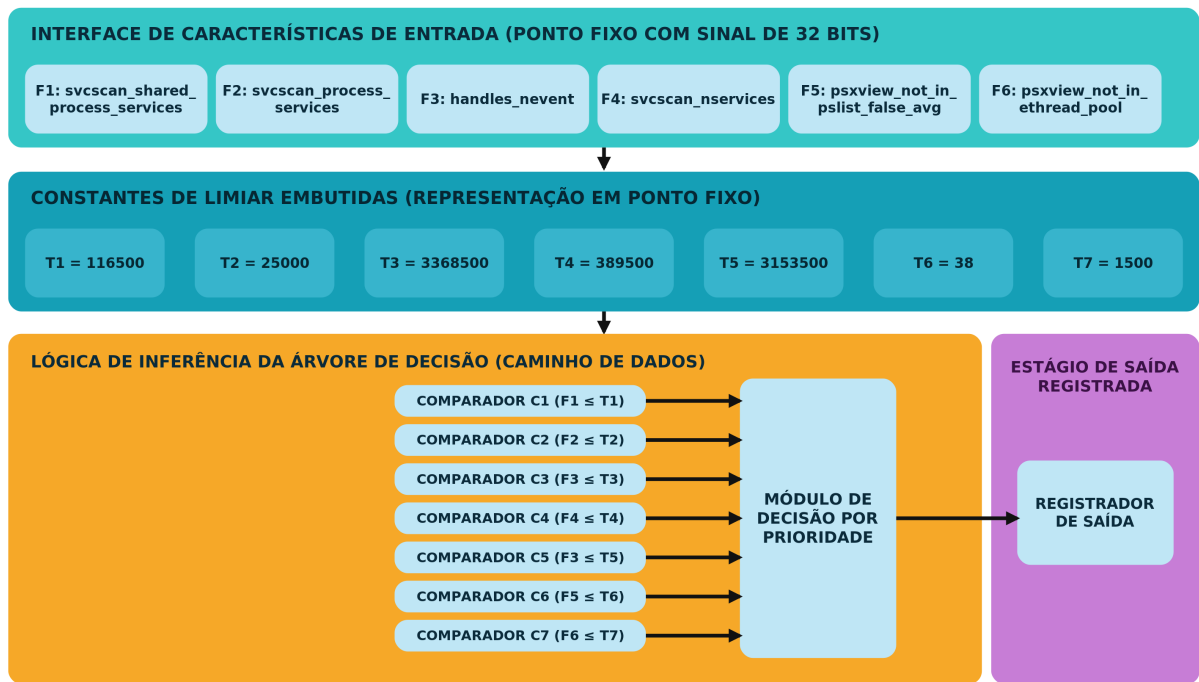
Regra da Árvore	Representação em <i>Hardware</i>	Mapeamento em FPGA
se característica $\leq 0,42$	Operação de comparação	Comparador em ponto fixo (LUT)
limiar = 0,42	Constante em ponto fixo	Constante embutida em VHDL
saída benigno / <i>ransomware</i>	Sinal de classificação binária	Lógica de saída registrada
caminho hierárquico de decisão	Cadeia de avaliação condicional	Caminho de dados combinacional
múltiplas avaliações de nós	Comparações paralelas	Paralelismo espacial no FPGA

Fonte: Elaborada pela autora (2026).

Os limiares do modelo são embutidos como constantes em ponto fixo, eliminando ope-

rações em ponto flutuante e lógica de controle iterativa. Cada característica de entrada é comparada ao seu limiar por um comparador dedicado, e os sinais resultantes são processados por um módulo de decisão que gera a classificação final. A Figura 18 detalha essa organização: os sete comparadores $C1$ a $C7$ avaliam diretamente cada característica de entrada contra o seu limiar em ponto fixo ($T1 = 116500$, $T2 = 25000$, $T3 = 3368500$, $T4 = 389500$, $T5 = 3153500$, $T6 = 38$ e $T7 = 1500$).

Figura 18 – Visão geral da arquitetura do classificador orientado a hardware em FPGA.



Fonte: Elaborada pela autora (2026).

Embora combinacional, a inferência permanece sujeita aos requisitos de temporização síncrona. A saída do classificador depende da propagação dos sinais ao longo do caminho de dados combinacional, composto pelos comparadores ($C1$ – $C7$), até o registrador de saída. Assim, o período de clock determina a vazão máxima, e as saídas dos comparadores, combinadas de acordo com a estrutura da árvore por um módulo de decisão por prioridade, produzem o sinal de classificação em um único ciclo.

A Tabela 4 resume a utilização de recursos no dispositivo-alvo (Intel Cyclone IV EP4CE115F29C7). O classificador consome apenas 74 elementos lógicos (menos de 1% dos 114.480 disponíveis), nenhum bloco de memória e nenhum multiplicador dedicado (DSP), confirmando o caráter leve da solução.

O consumo de potência, estimado após o place-and-route, resultou em dissipação térmica total de aproximadamente 176,6 mW (98,61 mW de potência estática do núcleo e 77,99 mW de potência de E/S). Considerando a latência de ciclo único de 20 ns a 50 MHz, a energia consumida por inferência é de aproximadamente 3,5 nJ.

Tabela 4 – Utilização de recursos de hardware no FPGA.

Recurso	Utilizado	Disponível	Utilização
Elementos lógicos	74	114.480	< 1%
Registradores	1	—	—
Pinos	195	529	37%
Bits de memória	0	3.981.312	0%
Multiplicadores (9 bits)	0	532	0%
PLLs	0	4	0%

Fonte: Elaborada pela autora (2026).

Importante destacar que, o atributo `handles.nevent`, por aparecer em dois nós da árvore, é avaliado por dois comparadores ($C3$ e $C5$), com limiares distintos. Os sinais resultantes são combinados por um módulo de decisão por prioridade que reproduz a estrutura hierárquica da árvore, o que leva à classificação final em um único ciclo. Embora empregado em dois nós com limiares distintos, sua repetição não implica, por si só, redundância. Entretanto, a inspeção da árvore revela que, no nó associado ao limiar $T5 = 3153500$, ambos os ramos conduzem à mesma classe, *ransomware*. Nesse caso específico, o resultado da comparação não altera a classificação final, permitindo que o comparador correspondente e a lógica de seleção associada sejam eliminados sem modificar o comportamento preditivo do modelo.

Nesta implementação, optou-se por preservar uma correspondência direta entre a árvore treinada e a arquitetura descrita em VHDL, o que facilita a rastreabilidade e a verificação funcional do circuito. Contudo, essa tradução estrutural não representa necessariamente a implementação mínima em termos de recursos. Trabalhos futuros podem explorar a poda de ramos com decisões equivalentes, a eliminação de comparadores redundantes e a simplificação das expressões booleanas do módulo de decisão. Tais otimizações podem reduzir a área ocupada na malha reconfigurável e, potencialmente, o caminho crítico e o consumo de potência, mantendo inalteradas as decisões produzidas pela árvore original.

Análise de Inferência

Após a implementação da arquitetura em FPGA, torna-se necessário avaliar seu comportamento durante a fase de inferência e compará-lo à execução do mesmo classificador em software. Enquanto os capítulos anteriores abordaram a seleção do modelo e sua materialização em hardware, este capítulo analisa os efeitos dessa implementação em termos de utilização de recursos, desempenho temporal e eficiência computacional.

A análise considera tanto a arquitetura sintetizada na FPGA quanto a implementação equivalente em software, utilizando exatamente o mesmo conjunto de dados e o mesmo modelo de classificação. Dessa forma, torna-se possível isolar os efeitos da plataforma de execução, evidenciando os ganhos obtidos com a implementação dedicada em hardware.

6.1 Análise Temporal da Inferência

A Tabela 5 apresenta o desempenho da inferência da Árvore de Decisão, implementada em software, para diferentes tamanhos de lote, considerando a latência média de processamento por lote, o tempo médio por amostra, o *throughput*, a utilização da CPU e o consumo de memória. Antes da realização das medições, foi executada uma etapa de *warm-up*, cujos resultados foram descartados, com o objetivo de reduzir os efeitos de inicialização do ambiente de execução e estabilizar as medições.

Tabela 5 – Métricas de Desempenho da Inferência de Software

Tamanho do lote	Latência média (s)	Tempo/Inst. (s)	Throughput (inst./s)	CPU (%)	Mem. (MB)
1	0,004100	0,0041000	244	87,38	336,43
50	0,004063	0,00008126	12 306	102,21	336,43
100	0,003489	0,00003489	28 662	103,45	336,43
150	0,002971	0,00001981	50 488	116,75	336,43
500	0,002346	0,000004692	213 129	84,79	336,74
1000	0,003564	0,000003564	280 584	79,26	337,28

Fonte: Elaborada pela autora (2026).

Ainda de acordo com a Tabela 5, observa-se que a latência média de processamento

do lote permaneceu na ordem de milissegundos para todos os tamanhos avaliados, variando entre aproximadamente 2,35 ms e 4,10 ms. À medida que o número de amostras processadas em conjunto aumenta, verifica-se uma redução do tempo médio por amostra, possivelmente decorrente da amortização dos custos fixos associados ao ambiente de execução. Como consequência, a vazão aumenta de aproximadamente 244 inferências por segundo, no processamento individual, para cerca de 281 mil inferências por segundo, quando o tamanho do lote é elevado para 1.000 amostras. Ressalta-se que, nos experimentos com lotes, o tempo médio por amostra foi obtido pela divisão da latência do lote pelo número de amostras processadas, o que não corresponde à latência efetivamente observada no processamento individual de cada amostra.

Em relação ao consumo de CPU, a utilização permaneceu elevada durante os experimentos, variando entre 79,26% e 116,75%. De acordo com a forma de contabilização adotada pela ferramenta de monitoramento, valores superiores a 100% indicam o uso simultâneo de mais de um núcleo de processamento. Esse comportamento evidencia que o desempenho da inferência em software é influenciado pelos recursos computacionais disponibilizados pela plataforma de execução. Em dispositivos com capacidade de processamento limitada, como os encontrados em ambientes de computação pervasiva, essa característica pode limitar a aplicabilidade e a escalabilidade da solução.

O consumo de memória apresentou comportamento praticamente constante ao longo dos experimentos, mantendo-se próximo de 337 MB. Esse resultado sugere que a maior parcela da memória utilizada está associada ao ambiente de execução da linguagem Python, às bibliotecas de AM e às estruturas internas do modelo, e é pouco influenciada pela quantidade de amostras processadas em cada lote. Esse consumo representa uma limitação relevante para abordagens executadas exclusivamente em software, especialmente quando destinadas a plataformas embarcadas e a dispositivos com recursos computacionais limitados.

A Tabela 6 apresenta uma comparação entre a execução do classificador em software e sua implementação em FPGA. No ambiente de software, o tempo médio por amostra variou entre 3,564 μ s, para lotes de 1.000 amostras, e 4,100 ms, para o processamento individual. A utilização da CPU variou entre aproximadamente 79% e 117%, enquanto o consumo de memória residente permaneceu próximo de 337 MB. Em contrapartida, a implementação em FPGA realiza a inferência em um único ciclo de *clock* de 20 ns, com latência determinística e sem utilizar blocos de memória dedicados. Para a comparação direta da latência de uma inferência individual, deve-se considerar o resultado obtido em software com lote unitário; os tempos médios por amostra obtidos com lotes maiores são mais adequados à análise de vazão.

Essa diferença de latência traduz-se em ganhos expressivos de velocidade. Para uma única amostra, o speedup do FPGA em relação ao software chega à ordem de 10^5 ; para lotes maiores, nos quais o software amortiza sua sobrecarga fixa, o ganho mantém-se na

Tabela 6 – Comparação de recursos: execução em software vs. FPGA.

Métrica	Software (CPU)	FPGA
Modelo de execução	CPU de propósito geral	Lógica dedicada
Latência média de inferência	3,564 us – 4,100 ms	20 ns
Utilização de CPU	79,26–116,75%	não se aplica
Memória residente	336,43–337,28 MB	sem blocos dedicados
Elementos lógicos	—	74
DSP / multiplicadores	—	0
Frequência de clock	dependente da CPU	50 MHz
Determinismo	dependente do SO/runtime	latência determinista

Fonte: Elaborada pela autora (2026).

ordem de 10^2 a 10^3 . Em termos de desempenho de classificação agregado sobre os lotes avaliados, a implementação em FPGA atinge 98,33% de acurácia, 99,00% de precisão e 97,55% de recall (Tabela 7). Esse valor é ligeiramente inferior à execução em software (cerca de 99,97% de acurácia), diferença atribuível ao uso de aritmética de ponto fixo em vez de ponto flutuante, mas mantém desempenho preditivo comparável a uma fração ínfima do custo de latência.

Tabela 7 – Comparação do desempenho da inferência entre a implementação em software e FPGA.

Implementação	Acurácia	Precisão	Recall	Latência (máx.)
Software	99,97%	99,94%	100,00%	4,1 ms
FPGA	98,33%	99,00%	97,55%	20 ns

Fonte: Elaborada pela autora (2026).

A Tabela 8 apresenta a comparação da latência de inferência entre as implementações em software e em FPGA para diferentes tamanhos de lote. Observa-se que a arquitetura em FPGA mantém um tempo de processamento constante de 20 ns por amostra, independentemente do número de instâncias processadas. Como a inferência é realizada em um único ciclo de clock, o tempo total de execução cresce linearmente com o número de amostras, atingindo apenas 20 μ s para um lote de mil instâncias.

O ganho de desempenho da implementação em FPGA foi quantificado por meio do *speedup*, definido como a razão entre o tempo de processamento em software e o tempo correspondente no classificador em hardware. Para uma única amostra, a latência de 4,100 ms em software, comparada ao ciclo de 20 ns da FPGA, resulta em um *speedup* de aproximadamente $2,05 \times 10^5$. Para lotes maiores, esse ganho diminui progressivamente, atingindo cerca de $1,78 \times 10^2$ para 1.000 amostras. Essa redução ocorre porque o processamento em lote permite amortizar os custos fixos da execução em software. Ressalta-se

Tabela 8 – Comparação da latência de inferência para diferentes cargas de trabalho.

Instâncias	Implementação	Latência/Amostra	Tempo Total	Speedup
1	Software FPGA	4,100 ms 20 ns	4,100 ms 20 ns	$1 \times$ $\approx 2,05 \times 10^5$
50	Software FPGA	81,26 μ s 20 ns	4,063 ms 1 μ s	$1 \times$ $\approx 4,06 \times 10^3$
100	Software FPGA	34,89 μ s 20 ns	3,489 ms 2 μ s	$1 \times$ $\approx 1,74 \times 10^3$
150	Software FPGA	19,81 μ s 20 ns	2,971 ms 3 μ s	$1 \times$ $\approx 9,90 \times 10^2$
500	Software FPGA	4,692 μ s 20 ns	2,346 ms 10 μ s	$1 \times$ $\approx 2,35 \times 10^2$
1000	Software FPGA	3,564 μ s 20 ns	3,564 ms 20 μ s	$1 \times$ $\approx 1,78 \times 10^2$

Fonte: Elaborada pela autora (2026).

que os valores da FPGA correspondem à latência do núcleo classificador e não incluem eventuais custos de aquisição, transferência ou preparação das características de entrada.

6.1.1 Impacto da Aritmética de Ponto Fixo no Desempenho de Classificação

Diferentemente do modelo em software, que opera sobre valores em ponto flutuante de precisão dupla, a arquitetura em FPGA representa tanto as características de entrada quanto os limiares aprendidos por meio de aritmética de ponto fixo com sinal, em palavras de 32 bits. Essa escolha elimina a necessidade de unidades aritméticas de ponto flutuante, cuja implementação demandaria número significativamente maior de elementos lógicos, múltiplos ciclos de *clock* por operação e, possivelmente, blocos DSP dedicados, sendo compatível com o objetivo de uma solução leve, de inferência em ciclo único e baixo consumo de recursos.

A conversão dos valores originais, em ponto flutuante, para a representação em ponto fixo foi realizada por meio de um fator de escala multiplicativo. Cada característica de entrada e cada limiar do modelo foram multiplicados por um fator de escala igual a 1.000 e, em seguida, truncados para a parte inteira, antes de sua representação binária em complemento de dois. Esse procedimento corresponde a uma representação de ponto fixo com três casas decimais de resolução, de modo que duas grandezas originalmente distintas apenas na quarta casa decimal, ou além dela, tornam-se indistinguíveis após a quantização.

A escolha do fator 1.000, e não de um valor de escala maior (por exemplo, 10.000

ou 100.000), decorreu de uma restrição prática imposta pela largura de palavra adotada. Como os limiares extraídos da Árvore de Decisão treinada já correspondem, em sua maioria, a grandezas absolutas dos atributos de *handles*, contagens de serviços e demais características do CIC-MalMem-2022, cujos valores nas colunas originais chegam a dezenas de milhares, a exemplo do limiar $T5 = 3153500$ na representação em *hardware*, correspondente a um valor original de 3153,5, o uso de um fator de escala maior aumentaria proporcionalmente a magnitude dos valores inteiros representados, elevando o risco de estouro (*overflow*) do intervalo representável em 32 bits com sinal ($[-2^{31}, 2^{31} - 1]$, aproximadamente $\pm 2,15 \times 10^9$) para os atributos de maior magnitude, ou exigindo, para evitar esse risco, o aumento da largura de palavra dos comparadores e, conseqüentemente, da área ocupada e do caminho crítico da arquitetura. O fator 1.000 representou, portanto, um compromisso entre a resolução necessária para preservar as fronteiras de decisão aprendidas pela árvore e a margem de segurança em relação ao limite de representação da palavra de 32 bits, sem exigir lógica adicional de saturação ou verificação de *overflow*.

Cabe destacar que a adequação de um fator de escala não pode ser avaliada isoladamente a partir de sua magnitude, mas sim em relação à distância entre os valores das características de teste e os limiares de decisão correspondentes: quando essa distância é da ordem de grandeza do passo de quantização (10^{-3} , no fator adotado) ou inferior a ele, a comparação em ponto fixo pode produzir um resultado diferente daquele obtido em ponto flutuante, alterando o ramo da árvore percorrido pela amostra e, potencialmente, sua classificação final. Esse mecanismo ajuda a explicar a origem da diferença de desempenho observada entre as duas implementações: a redução de acurácia de 99,97% para 98,33% (Tabela 7) concentra-se, presumivelmente, nas amostras cujas características de entrada situam-se muito próximas de um ou mais dos sete limiares de decisão ($T1$ a $T7$), e não distribuída uniformemente sobre o conjunto avaliado.

Essa concentração do erro nas proximidades das fronteiras de decisão também ajuda a explicar por que a queda no *recall* (de 100,00% em software para 97,55% em FPGA, uma diferença de 2,45 pontos percentuais) foi proporcionalmente maior que a queda na precisão (de 99,94% para 99,00%, uma diferença de 0,94 ponto percentual). Um deslocamento da fronteira de decisão causado pela quantização tende a produzir tanto falsos positivos quanto falsos negativos, mas, neste conjunto de dados e nesta árvore específica, o efeito líquido parece ter favorecido levemente a classe benigna, resultando em uma proporção maior de amostras de *ransomware* classificadas incorretamente como benignas. Do ponto de vista da aplicação de segurança, essa é a direção de erro mais indesejável, uma vez que um falso negativo representa a não detecção de uma ameaça ativa, ao passo que um falso positivo apenas gera uma verificação adicional. Ainda que a diferença observada seja pequena em termos absolutos, ela evidencia que a quantização em ponto fixo não é neutra em relação ao tipo de erro introduzido, e que a escolha do fator de escala deveria, idealmente, ser orientada não apenas pela margem de segurança em relação ao *overflow*,

mas também por uma análise de sensibilidade da distância entre os valores do conjunto de teste e cada um dos limiares da árvore.

Conclusão

Esta monografia apresentou o *design* e a implementação de uma arquitetura baseada em FPGA para a detecção de *ransomware* para ambientes de computação pervasiva. A arquitetura foi desenhada a partir da tradução de uma Árvore de Decisão treinada. A escolha do modelo foi sustentada por uma comparação sistemática entre cinco algoritmos de aprendizado de máquina e por uma análise de explicabilidade via SHAP, e a solução resultante realiza inferência determinista em um único ciclo por meio de uma lógica combinacional leve.

Retomando as questões de pesquisa formuladas na introdução, os resultados obtidos permitem apresentar as seguintes respostas. Quanto à questão (a), relativa ao modelo com a melhor relação entre desempenho preditivo e simplicidade estrutural na detecção de *ransomware* a partir do conjunto de dados CIC-MalMem-2022, a Árvore de Decisão mostrou-se a alternativa mais adequada. O modelo alcançou desempenho preditivo comparável ao dos métodos de *ensemble* Random Forest e XGBoost, com acurácia de 0,9997 e AUC-ROC próxima da unidade, apresentando, simultaneamente, a menor complexidade estrutural entre os modelos avaliados.

Em relação à questão (b), a análise baseada em SHAP revelou uma convergência parcial entre os atributos mais relevantes para modelos de naturezas distintas, com predominância de características relacionadas aos serviços do sistema. Esse resultado indica que os modelos avaliados sustentam parte significativa de suas predições com base em um conjunto semelhante de atributos e fornece evidências de sua capacidade discriminativa no conjunto de dados analisado. A análise também mostrou que a Árvore de Decisão utiliza o menor número de características entre os modelos avaliados. Ressalta-se, entretanto, que a convergência observada não permite, isoladamente, descartar a existência de correlações sem efeito, uma vez que os valores SHAP explicam a influência dos atributos sobre as predições, mas não estabelecem relações de causalidade.

Quanto à questão (c), verificou-se que o classificador pode ser implementado como uma arquitetura predominantemente combinacional em FPGA, ocupando 74 elementos lógicos, menos de 1% do dispositivo, sem utilizar multiplicadores dedicados ou blocos de

memória. Neste contexto, pode-se pensar em vários módulos classificadores na mesma malha reconfigurável para prover uma solução de detecção distribuída e em paralelo.

A implementação realiza cada inferência em um ciclo de *clock* de 20 ns, enquanto o tempo por amostra em software variou entre 3,564 μ s, para lotes de 1.000 amostras, e 4,100 ms, para o processamento individual. Portanto, os resultados confirmam a redução do tempo de inferência em relação à execução no software. Em contrapartida, a acurácia foi reduzida de 99,97% no modelo em software para 98,33% na implementação em FPGA, o que corresponde a uma diferença de 1,64 pontos percentuais.

Embora a implementação tenha preservado um desempenho preditivo elevado, a classificação dessa diferença como significativa ou não significativa depende da definição de um critério de aceitabilidade ou da realização de uma análise estatística específica.

7.1 Trabalhos Futuros

Como desdobramentos deste trabalho, pretende-se avaliar abordagens baseadas em *ensembles*, como *Random Forest* e *Gradient Boosting*, em arquiteturas de FPGA, investigando os *trade-offs* entre acurácia preditiva, complexidade de hardware, utilização de recursos e latência de inferência. Também se vislumbra a integração do classificador a um *pipeline* completo de monitoramento contínuo, contemplando a extração de características em tempo real e a sincronização com eventos do sistema operacional. Nesse cenário, o uso de processadores híbridos (e.g., Zynq), que combinam um núcleo de processamento de propósito geral (*Processing System*) com lógica programável (*Programmable Logic*) no mesmo dispositivo, permitiria que a extração de características a partir da memória do sistema monitorado fosse conduzida em software, enquanto a classificação seria delegada ao núcleo em FPGA aqui proposto.

Um dos desafios associados a essa integração é o tratamento de variações dinâmicas de carga, ou seja, da variação, ao longo do tempo, na taxa com que novas amostras (processos, módulos carregados ou eventos do sistema monitorado) precisam ser submetidas ao classificador. Em ambientes de computação pervasiva, essa taxa não é constante: picos de criação de processos, execução simultânea de múltiplas aplicações ou intervalos de amostragem variáveis do subsistema de monitoramento podem gerar rajadas de amostras que excedem, momentaneamente, a capacidade de processamento sequencial de uma única instância do classificador, mesmo esta operando em um único ciclo de *clock* por inferência. Como trabalho futuro, pretende-se investigar estratégias de tratamento dessas rajadas, como a replicação de múltiplas instâncias do classificador combinacional operando em paralelo sobre a mesma malha reconfigurável, mecanismos de enfileiramento (*buffering*) das características extraídas entre o núcleo de processamento e a lógica de classificação, e políticas de escalonamento que priorizem a classificação de processos recém-criados ou de maior risco, de modo a evitar o descarte ou o atraso excessivo de amostras sob carga

elevada, sem comprometer a natureza determinista e de baixa latência da arquitetura proposta.

Por fim, vislumbra-se ainda a investigação de mecanismos de reconfiguração parcial para a atualização do modelo em campo, sem interrupção do serviço.

Referências

- ALCOLEA, A.; RESANO, J. FPGA accelerator for gradient boosting decision trees. **Electronics**, MDPI, v. 10, n. 3, p. 314, 2021. Citado 4 vezes nas páginas 12, 14, 21 e 22.
- ALRAIZZA, A.; ALGARNI, A. Ransomware detection using machine learning: A survey. **Big Data and Cognitive Computing**, MDPI, v. 7, n. 3, p. 143, 2023. Citado 3 vezes nas páginas 11, 17 e 18.
- ALRAWASHDEH, K.; PURDY, C. Fast hardware assisted online learning using unsupervised deep learning structure for anomaly detection. In: **International Conference on Information and Computer Technologies (ICICT)**. New York: IEEE, 2018. p. 128–134. Citado 2 vezes nas páginas 11 e 20.
- ALWASHALI, A. A. M. A.; RAHMAN, N. A. A.; ISMAIL, N. A survey of ransomware as a service (RaaS) and methods to mitigate the attack. In: **14th International Conference on Developments in eSystems Engineering (DeSE)**. New York: IEEE, 2021. p. 92–96. Citado na página 17.
- ALZAHRANI, S. et al. A survey of ransomware detection methods. **IEEE Access**, IEEE, 2025. Citado na página 17.
- ANAND, P. M.; CHARAN, P. V. S.; SHUKLA, S. K. HiPeR - early detection of a ransomware attack using hardware performance counters. **Digital Threats: Research and Practice**, ACM, v. 4, n. 3, p. 1–24, 2023. Citado 3 vezes nas páginas 12, 21 e 22.
- ARP, D. et al. Dos and don'ts of machine learning in computer security. In: **31st USENIX Security Symposium**. Berkeley, CA: USENIX Association, 2022. p. 3971–3988. Citado 2 vezes nas páginas 13 e 20.
- AZMOODEH, A. et al. Detecting crypto-ransomware in IoT networks based on energy consumption footprint. **Journal of Ambient Intelligence and Humanized Computing**, Springer, v. 9, n. 4, p. 1141–1152, 2018. Citado na página 16.
- BREIMAN, L. Random forests. **Machine Learning**, Springer, v. 45, n. 1, p. 5–32, 2001. Citado na página 18.
- BREIMAN, L. et al. **Classification and Regression Trees**. Belmont, CA: Wadsworth, 1984. Citado na página 18.

- Canadian Institute for Cybersecurity. **CIC-MalMem-2022 Dataset**. 2022. <<https://www.unb.ca/cic/datasets/malmem-2022.html>>. Acesso em: 10 out. 2025. Citado na página 23.
- CARRIER, T. et al. Detecting obfuscated malware using memory feature engineering. In: **Proceedings of the 8th International Conference on Information Systems Security and Privacy (ICISSP)**. Setúbal, Portugal: SciTePress, 2022. p. 177–188. Citado 4 vezes nas páginas 11, 13, 17 e 23.
- CHEN, T.; GUESTRIN, C. XGBoost: A scalable tree boosting system. In: **Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York: ACM, 2016. p. 785–794. Citado na página 19.
- CORTES, C.; VAPNIK, V. Support-vector networks. **Machine Learning**, Springer, v. 20, n. 3, p. 273–297, 1995. Citado na página 19.
- CRUVINEL, C. et al. An fpga-accelerated decision tree model for ransomware detection in pervasive environments. In: **Anais do XVIII Simpósio Brasileiro de Computação Ubíqua e Pervasiva**. Porto Alegre, RS, Brasil: SBC, 2026. p. 46–57. ISSN 2595-6183. Disponível em: <<https://sol.sbc.org.br/index.php/sbcup/article/view/43286>>. Citado na página 14.
- GAJJAR, A. et al. RD-FAXID: Ransomware detection with FPGA-accelerated XGBoost. **ACM Transactions on Reconfigurable Technology and Systems**, ACM, v. 17, n. 4, p. 1–33, 2024. Citado 5 vezes nas páginas 12, 14, 20, 21 e 22.
- HAQUE, M.; AHAMED, S. I. Security in pervasive computing: Current status and open issues. **International Journal of Network Security**, v. 3, n. 3, p. 203–214, 2006. Citado na página 16.
- Hornetsecurity. **Ransomware Attacks Survey 2022**. 2022. <<https://www.hornetsecurity.com>>. Acesso em: 10 out. 2025. Citado na página 11.
- HUMAYUN, M. et al. Internet of things and ransomware: Evolution, mitigation and prevention. **Egyptian Informatics Journal**, Elsevier, v. 22, n. 1, p. 105–117, 2021. Citado 2 vezes nas páginas 11 e 17.
- KULAGA, R.; GORGON, M. FPGA implementation of decision trees and tree ensembles for character recognition in Vivado HLS. In: **Image Processing and Communications Challenges**. Cham: Springer, 2014. Citado 4 vezes nas páginas 12, 14, 21 e 22.
- LECUN, Y.; BENGIO, Y.; HINTON, G. Deep learning. **Nature**, Nature Publishing Group, v. 521, n. 7553, p. 436–444, 2015. Citado na página 19.
- LIPOVSKÝ, R.; STEFANKO, L.; BRANISA, G. **The rise of Android ransomware**. Bratislava, 2016. Citado na página 17.
- LUNDBERG, S. M.; LEE, S.-I. A unified approach to interpreting model predictions. In: **Advances in Neural Information Processing Systems (NeurIPS)**. Red Hook, NY: Curran Associates, 2017. v. 30, p. 4765–4774. Citado 2 vezes nas páginas 12 e 19.

- MAKRANI, H. M. et al. A comprehensive memory analysis of data-intensive workloads on reconfigurable hardware. **ACM Transactions on Reconfigurable Technology and Systems**, ACM, 2022. Citado na página 20.
- MALIK, S.; KHANNA, A.; SHARMA, D. Early-stage ransomware detection: a comprehensive review. **Journal of Information Security and Applications**, 2024. Citado 2 vezes nas páginas 11 e 17.
- MOHURLE, S.; PATIL, M. A brief study of WannaCry threat: Ransomware attack 2017. **International Journal of Advanced Research in Computer Science**, v. 8, n. 5, 2017. Citado na página 17.
- NGO, D.-M. et al. FPGA hardware acceleration framework for anomaly-based intrusion detection system in IoT. In: **31st International Conference on Field-Programmable Logic and Applications (FPL)**. New York: IEEE, 2021. p. 69–75. Citado 2 vezes nas páginas 12 e 20.
- OZ, H. et al. A survey on ransomware: Evolution, taxonomy, and defense solutions. **ACM Computing Surveys**, ACM, v. 54, n. 11s, p. 1–37, 2022. Citado na página 17.
- PARK, J. H. et al. Ransomware-based cyber attacks: A comprehensive survey. **Journal of Internet Technology**, v. 23, n. 7, p. 1557–1564, 2022. Citado na página 16.
- SATYANARAYANAN, M. Pervasive computing: Vision and challenges. **IEEE Personal Communications**, v. 8, n. 4, p. 10–17, 2002. Citado na página 16.
- SHAPLEY, L. S. A value for n-person games. **Contributions to the Theory of Games**, v. 2, n. 28, p. 307–317, 1953. Citado na página 19.

Apêndices

APÊNDICE **A**

Códigos-Fonte Desenvolvidos

O código a seguir apresenta a arquitetura desenvolvida em VHDL para validar o funcionamento desta proposta.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity avr_decisao is
  Port (
    clk    : in STD_LOGIC;
    reset  : in STD_LOGIC;

    -- Entradas em ponto fixo, multiplicadas por 1000.
    -- Exemplo: 116.5 = 116500
    svcscan_shared_process_services : in SIGNED(31 downto 0);
    svcscan_process_services        : in SIGNED(31 downto 0);
    handles_nevent                   : in SIGNED(31 downto 0);
    svcscan_nservices                : in SIGNED(31 downto 0);

    -- Exemplo: 0.038 = 38
    psxview_not_in_pslist_false_avg : in SIGNED(31 downto 0);

    -- Exemplo: 1.5 = 1500
    psxview_not_in_ethread_pool_false_avg : in SIGNED(31 downto 0);

    -- '1' = ransomware
    -- '0' = benigno
    is_ransomware : out STD_LOGIC
  );
end entity;
```

```
    );
end avr_decisao;

architecture Behavioral of avr_decisao is

    constant SHARED_PROCESS_THRESHOLD : SIGNED(31 downto 0) :=
        to_signed(116500, 32); -- 116.5

    constant PROCESS_SERVICES_THRESHOLD : SIGNED(31 downto 0) :=
        to_signed(25000, 32); -- 25.0

    constant HANDLES_NEVENT_THRESHOLD_1 : SIGNED(31 downto 0) :=
        to_signed(3368500, 32); -- 3368.5

    constant NSERVICES_THRESHOLD : SIGNED(31 downto 0) :=
        to_signed(389500, 32); -- 389.5

    constant HANDLES_NEVENT_THRESHOLD_2 : SIGNED(31 downto 0) :=
        to_signed(3153500, 32); -- 3153.5

    constant PSXLIST_THRESHOLD : SIGNED(31 downto 0) :=
        to_signed(38, 32); -- 0.038

    constant ETHREAD_POOL_THRESHOLD : SIGNED(31 downto 0) :=
        to_signed(1500, 32); -- 1.5

begin

    detection_process : process(clk, reset)
    begin
        if reset = '1' then
            is_ransomware <= '0';

        elsif rising_edge(clk) then

            -- Nó raiz:
            -- svcscan_shared_process_services <= 116.5
            if svcscan_shared_process_services <=
                SHARED_PROCESS_THRESHOLD then
```

```
-- Segundo nível:
-- svcscan_process_services <= 25.0
if svcscan_process_services <=
    PROCESS_SERVICES_THRESHOLD then

    -- Terceiro nível:
    -- handles_nevent <= 3368.5
    if handles_nevent <=
        HANDLES_NEVENT_THRESHOLD_1 then

        -- Quarto nível, ramo esquerdo:
        -- svcscan_nservices <= 389.5
        if svcscan_nservices <=
            NSERVICES_THRESHOLD then

            -- Quinto nível:
            -- handles_nevent <= 3153.5
            if handles_nevent <=
                HANDLES_NEVENT_THRESHOLD_2 then
                is_ransomware <= '1';
            else
                is_ransomware <= '1';
            end if;

        else

            -- Quinto nível:
            -- psxview_not_in_pslist_false_avg <= 0.038
            if psxview_not_in_pslist_false_avg <=
                PSXLIST_THRESHOLD then
                is_ransomware <= '1';
            else
                is_ransomware <= '0';
            end if;

        end if;

    else

end if;
```

```
-- Quarto nível, ramo direito:
-- psxview_not_in_ethread_pool_false_avg <= 1.5
if psxview_not_in_ethread_pool_false_avg <=
    ETHREAD_POOL_THRESHOLD then
    is_ransomware <= '1';
else
    is_ransomware <= '0';
end if;

end if;

else
    -- svcsan_process_services > 25.0
    is_ransomware <= '0';
end if;

else
    -- svcsan_shared_process_services > 116.5
    is_ransomware <= '0';
end if;

end if;
end process detection_process;

end Behavioral;
```

O código a seguir apresenta o exemplo de testbench desenvolvido em VHDL para validar o funcionamento da arquitetura proposta, por meio da aplicação de diferentes casos de teste e da verificação automática dos resultados.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity tb_arv_decisao is
end tb_arv_decisao;

architecture Behavioral of tb_arv_decisao is
```

```
signal clk    : STD_LOGIC := '0';
signal reset : STD_LOGIC := '0';

-- Entradas da árvore em ponto fixo, multiplicadas por 1000
signal svcscan_shared_process_services :
    SIGNED(31 downto 0) := (others => '0');

signal svcscan_process_services :
    SIGNED(31 downto 0) := (others => '0');

signal handles_nevent :
    SIGNED(31 downto 0) := (others => '0');

signal svcscan_nservices :
    SIGNED(31 downto 0) := (others => '0');

signal psxview_not_in_pslist_false_avg :
    SIGNED(31 downto 0) := (others => '0');

signal psxview_not_in_ethread_pool_false_avg :
    SIGNED(31 downto 0) := (others => '0');

signal is_ransomware : STD_LOGIC;

constant CLK_PERIOD : TIME := 10 ns;

begin

-----
-- Dispositivo sob teste
-----

DUT : entity work.avr_decisao
    port map (
        clk    => clk,
        reset => reset,

        svcscan_shared_process_services =>
            svcscan_shared_process_services,
```

```
        svcscan_process_services =>
            svcscan_process_services,

        handles_nevent =>
            handles_nevent,

        svcscan_nservices =>
            svcscan_nservices,

        psxview_not_in_pslist_false_avg =>
            psxview_not_in_pslist_false_avg,

        psxview_not_in_ethread_pool_false_avg =>
            psxview_not_in_ethread_pool_false_avg,

        is_ransomware =>
            is_ransomware
    );

-----
-- Geração do clock
-----

clk_process : process
begin
    while true loop
        clk <= '0';
        wait for CLK_PERIOD / 2;

        clk <= '1';
        wait for CLK_PERIOD / 2;
    end loop;
end process clk_process;

-----
-- Processo de estímulos
-----

test_process : process

    -- Executa um teste completo.
```

```
--
-- Os valores recebidos por essa procedure já devem estar
-- multiplicados por 1000.
procedure run_test (
    constant test_name : in STRING;
    constant f1         : in INTEGER;
    constant f2         : in INTEGER;
    constant f3         : in INTEGER;
    constant f4         : in INTEGER;
    constant f5         : in INTEGER;
    constant f6         : in INTEGER;
    constant expected   : in STD_LOGIC
) is
begin
    report test_name severity note;

    svcscan_shared_process_services <= to_signed(f1, 32);
    svcscan_process_services        <= to_signed(f2, 32);
    handles_nevent                  <= to_signed(f3, 32);
    svcscan_nservices               <= to_signed(f4, 32);

    psxview_not_in_pslist_false_avg <=
        to_signed(f5, 32);

    psxview_not_in_ethread_pool_false_avg <=
        to_signed(f6, 32);

    -- O DUT registra a saída na borda de subida.
    wait until rising_edge(clk);

    -- Aguarda a atualização do sinal após o delta cycle.
    wait for 1 ns;

    assert is_ransomware = expected
        report
            "Falha em " & test_name &
            ". Esperado = " &
            STD_LOGIC'image(expected) &
            ", obtido = " &
```

```
        STD_LOGIC'image(is_ransomware)
        severity error;
end procedure run_test;

begin

-----
-- Reset assíncrono
-----

report "Aplicando reset" severity note;

reset <= '1';
wait for 2 * CLK_PERIOD;

assert is_ransomware = '0'
    report "Erro: a saída não foi zerada durante o reset"
    severity error;

reset <= '0';
wait until falling_edge(clk);

-----
-- Teste 1
-- C1: F1 > T1
-- Resultado: benigno
-----

run_test(
    test_name => "Teste 1: F1 maior que T1",
    f1        => 117000,    -- 117.0 > 116.5
    f2        => 20000,
    f3        => 3000000,
    f4        => 300000,
    f5        => 20,
    f6        => 1000,
    expected  => '0'
);

-----
-- Teste 2
```

```
-- C1 verdadeiro, C2 falso: F2 > T2
```

```
-- Resultado: benigno
```

```
-----  
run_test(  
    test_name => "Teste 2: F2 maior que T2",  
    f1        => 100000,    -- 100.0 <= 116.5  
    f2        => 26000,    -- 26.0 > 25.0  
    f3        => 3000000,  
    f4        => 300000,  
    f5        => 20,  
    f6        => 1000,  
    expected  => '0'  
);
```

```
-----  
-- Teste 3
```

```
-- C1, C2, C3, C4 e C5 verdadeiros
```

```
-- Resultado: ransomware  
-----
```

```
run_test(  
    test_name => "Teste 3: ramo C5 verdadeiro",  
    f1        => 100000,    -- <= 116.5  
    f2        => 20000,    -- <= 25.0  
    f3        => 3000000,    -- <= 3368.5 e <= 3153.5  
    f4        => 300000,    -- <= 389.5  
    f5        => 50,  
    f6        => 2000,  
    expected  => '1'  
);
```

```
-----  
-- Teste 4
```

```
-- C1, C2, C3 e C4 verdadeiros; C5 falso
```

```
-- Os dois ramos de C5 classificam como ransomware.  
-----
```

```
run_test(  
    test_name => "Teste 4: ramo C5 falso",  
    f1        => 100000,  
    f2        => 20000,
```

```
f3      => 3200000,    -- > 3153.5 e <= 3368.5
f4      => 300000,     -- <= 389.5
f5      => 50,
f6      => 2000,
expected => '1'
);
```

```
-----
-- Teste 5
-- C3 verdadeiro, C4 falso e C6 verdadeiro
-- Resultado: ransomware
-----
```

```
run_test(
  test_name => "Teste 5: F5 menor ou igual a T6",
  f1        => 100000,
  f2        => 20000,
  f3        => 3200000,    -- <= 3368.5
  f4        => 400000,     -- > 389.5
  f5        => 30,         -- 0.030 <= 0.038
  f6        => 2000,
  expected  => '1'
);
```

```
-----
-- Teste 6
-- C3 verdadeiro, C4 falso e C6 falso
-- Resultado: benigno
-----
```

```
run_test(
  test_name => "Teste 6: F5 maior que T6",
  f1        => 100000,
  f2        => 20000,
  f3        => 3200000,
  f4        => 400000,
  f5        => 39,         -- 0.039 > 0.038
  f6        => 2000,
  expected  => '0'
);
```

```
-----  
-- Teste 7  
-- C3 falso e C7 verdadeiro  
-- Resultado: ransomware  
-----  
  
run_test(  
    test_name => "Teste 7: F3 alto e F6 menor ou igual a T7",  
    f1        => 100000,  
    f2        => 20000,  
    f3        => 3400000,    -- > 3368.5  
    f4        => 400000,  
    f5        => 50,  
    f6        => 1400,        -- 1.4 <= 1.5  
    expected  => '1'  
);
```

```
-----  
-- Teste 8  
-- C3 falso e C7 falso  
-- Resultado: benigno  
-----  
  
run_test(  
    test_name => "Teste 8: F3 alto e F6 maior que T7",  
    f1        => 100000,  
    f2        => 20000,  
    f3        => 3400000,  
    f4        => 400000,  
    f5        => 50,  
    f6        => 1600,        -- 1.6 > 1.5  
    expected  => '0'  
);
```

```
-----  
-- Teste 9  
-- Valores exatamente nos thresholds: comparações usam <=.  
-----  
  
run_test(  
    test_name => "Teste 9: valores exatamente nos thresholds",  
    f1        => 116500,
```

```
        f2          => 25000,
        f3          => 3368500,
        f4          => 389500,
        f5          => 38,
        f6          => 1500,
        expected    => '1'
    );

    report "Todos os testes foram concluídos com sucesso!"
        severity note;

    wait;
end process test_process;

end Behavioral;
```