



**UNIVERSIDADE FEDERAL DO AMAZONAS
FACULDADE DE TECNOLOGIA
ENGENHARIA ELÉTRICA**

NOME COMPLETO AQUI

ESCREVA O TÍTULO DO TRAVALHO

**MANAUS - AMAZONAS - BRASIL
2025**

NOME COMPLETO AQUI

ESCREVA O TÍTULO DO TRAVALHO

Relatório completo das atividades de iniciação científica apresentado à Pró-Reitoria de Pesquisa e Pós-Graduação, em cumprimento às exigências legais para finalização das ações do Programa de Institucional de Bolsas de Iniciação Científica 2024 - 2025.

Área de concentração: Engenharia XXXX

Orientador: Prof. Dr. Iury Valente de Bessa

**MANAUS - AMAZONAS - BRASIL
2025**

RESUMO

A regra do impedimento é uma das mais polêmicas existentes num jogo de futebol, porém não é comum a estudar a fundo sobre a mesma e discutir todos os cenários em que o impedimento se configura, o mais conhecido é o cenário do atacante contra o penúltimo defensor, é comum se pensar que precisa de apenas um defensor para que o adversário não fique impedido, mas como em quase todas as situações de jogo o goleiro está debaixo de suas traves, é fácil se esquecer que ele também conta como defensor e para a regra, outro cenário é referente à linha da bola, quando o atacante já passou os 2 últimos defensores, o mesmo pode realizar um passe para um companheiro que está na mesma linha ou atrás da linha da bola e por último, o impedimento primeiro se configura na linha de meio campo, se um jogador rumar ao ataque e se encontra antes da linha do meio campo, qualquer passe em sua direção pode ser realizado, entretanto, se os 2 últimos defensores estiverem no campo de ataque e o atacante estiver após a linha do meio de campo, qualquer bola direcionada ao mesmo será considerado impedimento. A leitura e base utilizada para este trabalho serão de pesquisas consolidadas e que já conseguiram reproduzir a proposta com sucesso ou que serviram de base para outros trabalhos na área e tem validade para este.

Palavras-chave: Arbitragem esportiva; Impedimento; Inteligência Artificial; Processamento Digital; Visão Computacional.

ABSTRACT

The offside rule is one of the most controversial in a football game, but it is not common to study the background of it and discuss the scenarios in which offside occurs. The best known is the scenario of the attacker against the penultimate defender. It is common to think that only one defender is needed to prevent the opponent from being offside. However, since in almost all game situations the goalkeeper is under his goalposts, it is easy to forget that he also counts as a defender and for the rule. Another scenario concerns the ball line. When the attacker has already passed the last 2 defenders, he can make a pass to a teammate who is on the same line or behind the ball line. Finally, offside first occurs on the halfway line. If a player heads to attack and is before the halfway line, any pass in his direction can be made. However, if the last 2 defenders are in the attacking field and the attacker is after the halfway line, any ball directed at him will be considered offside. The reading and basis used for this work will be consolidated research that has already managed to successfully reproduce the proposal or that served as a basis for other works in the area and are valid for this one.

Key-words: Sports arbitration; Offside; Artificial Intelligence; Digital Processing; Computer Vision.

LISTA DE ILUSTRAÇÕES

Figura 1 – Pipeline de verificação formal com o ESBMC.	8
Figura 2 – Tempo de verificação vs dimensão da matriz (GEMM com tiling).	12
Figura 3 – Tempo de verificação por iteração do agente neuro-simbólico.	13

SUMÁRIO

1	INTRODUÇÃO	6
2	OBJETIVOS	7
2.1	Geral	7
2.2	Específicos	7
3	METODOLOGIA	8
3.1	Visão Geral da Abordagem	8
3.2	Caso 1 — Verificação Direta de Modelos Python (Frontend Experimental)	9
3.3	Caso 2 — Verificação do Motor de Inferência (Kernels C++)	9
3.4	Caso 3 — Loop Neuro-Simbólico (Agente LLM + ESBMC)	9
3.5	Caso 4 — Controlador PID com Injeção de Caos	10
4	RESULTADO PARCIAL	11
4.1	Caso 1 — Verificação de Modelo Python	11
4.2	Caso 2 — Verificação de Kernels de Inferência	11
4.2.1	<i>Kernel de Atenção</i>	11
4.2.2	<i>Escalabilidade do GEMM</i>	11
4.3	Caso 3 — Loop Neuro-Simbólico	12
4.4	Caso 4 — Controlador PID com Engenharia do Caos	13
4.5	Análise Comparativa	13
5	CONCLUSÃO	15
	REFERÊNCIAS	16

1 INTRODUÇÃO

(Contendo a fundamentação teórica do tema abordado, o objeto do estudo e a justificativa para o estudo.)

2 OBJETIVOS

2.1 Geral

O objetivo principal deste projeto é...

2.2 Específicos

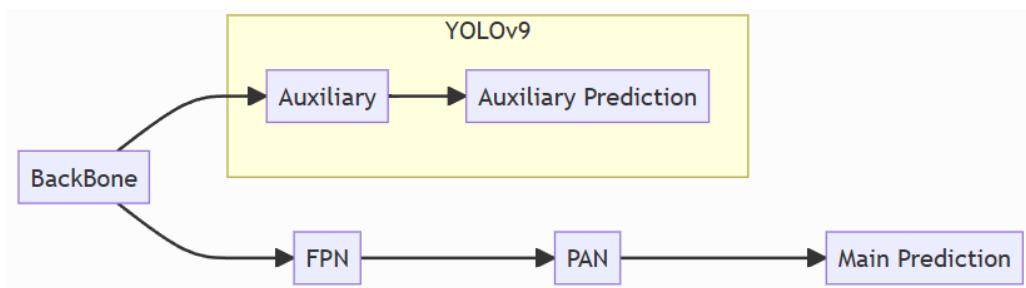
3 METODOLOGIA

Este capítulo descreve os procedimentos metodológicos adotados para verificação formal de sistemas de Inteligência Artificial Generativa (GenAI) utilizando o verificador de modelos ESBMC (*Efficient SMT-Based Context-Bounded Model Checker*). A investigação foi estruturada em quatro estudos de caso complementares, cada um explorando uma camada distinta do ecossistema de IA Generativa.

3.1 Visão Geral da Abordagem

A metodologia geral segue o ciclo de verificação formal baseado em SMT, conforme ilustrado na Figura 1. O ESBMC opera convertendo o código-fonte para uma representação intermediária (*GOTO-Programs*), realizando execução simbólica e codificando as restrições em fórmulas SMT (*Satisfiability Modulo Theories*). O solver SMT (Z3 ou Bitwuzla) então determina se existe uma atribuição de valores de entrada capaz de violar as propriedades especificadas — produzindo um contra-exemplo em caso afirmativo, ou uma prova de corretude dentro dos limites verificados.

Figura 1 – Pipeline de verificação formal com o ESBMC.



Fonte: Elaborada pelo autor, 2025.

Para cada caso de uso, foram definidas:

- Pré-condições:** restrições sobre os valores de entrada (ex.: intervalo de valores, tamanho de buffers);
- Propriedades de Segurança:** asserções formais que o sistema deve satisfazer para todo valor de entrada válido;
- Métricas de Avaliação:** tempo de verificação, escalabilidade e eficácia na detecção de falhas.

3.2 Caso 1 — Verificação Direta de Modelos Python (Frontend Experimental)

O primeiro caso de uso explorou o *frontend* experimental do ESBMC para código Python. Um neurônio artificial com função de ativação ReLU foi implementado em `neuron.py` com anotações de tipo estático, requisito obrigatório para que o ESBMC consiga inferir os tipos durante a análise.

As entradas $x_1, x_2 \in [0.0, 1.0]$ foram tratadas como variáveis simbólicas não-determinísticas, e as seguintes propriedades foram verificadas formalmente:

- a) Saída não-negativa: $\text{out} \geq 0.0$ (propriedade da ReLU);
- b) Saída limitada superiormente: $\text{out} \leq 0.6$ (limite teórico dado os pesos fixos).

O ESBMC foi executado com os flags `--floatbv --k-induction` para habilitar aritmética de ponto flutuante e indução-k.

3.3 Caso 2 — Verificação do Motor de Inferência (Kernels C++)

O segundo caso focou na verificação de segurança de memória em kernels C++ análogos aos utilizados em motores de inferência como `llama.cpp`. Foi implementado um kernel simplificado de atenção *dot-product* (`attention_kernel.cpp`) com alocação dinâmica de memória e laços de acesso a arrays.

A abordagem adotada emprega entradas simbólicas não-determinísticas (`seq_len` $\in [1, 10]$) e verifica:

- a) Ausência de *buffer overflows* nos acessos `query[i]` e `key[i]`;
- b) Ausência de vazamentos de memória (*memory leaks*).

Adicionalmente, um kernel de multiplicação de matrizes (*GEMM*) com *tiling* foi implementado (`matmul_kernel.cpp`) para análise de escalabilidade, variando a dimensão $N \in \{2, 3, 4, 5, 6\}$.

O comando utilizado foi: `esbmc attention_kernel.cpp --multi-property --memory-leak --overflow-check`.

3.4 Caso 3 — Loop Neuro-Simbólico (Agente LLM + ESBMC)

O terceiro caso simulou um ciclo de verificação contínua voltado a código gerado por modelos de linguagem (LLMs). O script `mock_agent.py` implementa um agente que itera sobre versões de código C fornecidas por um LLM simulado e submete cada versão ao ESBMC automaticamente.

O fluxo do agente é:

- a) **Iteração 0:** código com `strcpy` em buffer de tamanho fixo — vulnerável a *overflow*;
- b) **Iteração 1:** código corrigido usando `strncpy` — verificado com sucesso.

O tempo de verificação por iteração foi medido para avaliar o *overhead* introduzido pela integração ESBMC-LLM.

3.5 Caso 4 — Controlador PID com Injeção de Caos

O quarto caso aplicou princípios de *Engenharia do Caos* à verificação de sistemas de controle críticos. Um controlador PID digital (`pid_controller.c`) responsável por regular a temperatura de uma planta térmica foi submetido à injeção de ruído não-determinístico nos sensores (± 5.0 graus) a cada iteração.

A propriedade de segurança verificada foi: $T_{\text{sistema}} < T_{\text{MAX}} = 150.0^{\circ}\text{C}$ para todos os valores de ruído possíveis ao longo de 10 passos de simulação. O verificador realizou análise por limitação de passos (*bounded model checking*) com desenrolamento de 10 iterações do laço de controle.

4 RESULTADO PARCIAL

Este capítulo apresenta os resultados parciais obtidos, organizados por caso de uso. Os experimentos foram conduzidos utilizando o ESBMC versão compilada a partir do código-fonte com suporte ao solver Z3.

4.1 Caso 1 — Verificação de Modelo Python

A verificação formal do neurônio com ativação ReLU (`neuron.py`) resultou em **VERIFICAÇÃO BEM-SUCEDIDA** para ambas as propriedades especificadas. O ESBMC provou matematicamente, via k-indução, que para qualquer entrada $x_1, x_2 \in [0.0, 1.0]$, a saída do neurônio satisfaz $\text{out} \geq 0.0$ e $\text{out} \leq 0.6$.

O principal desafio identificado foi a exigência de tipagem estática rigorosa no *frontend* Python do ESBMC. Construções dinâmicas comuns em *frameworks* de ML — como listas heterogêneas e funções de ordem superior — exigem refatoração substancial antes de serem verificáveis, limitando a aplicabilidade direta a modelos já treinados com PyTorch ou TensorFlow.

Os resultados do log de verificação foram armazenados em `results/case1_mlp.log`.

4.2 Caso 2 — Verificação de Kernels de Inferência

4.2.1 Kernel de Atenção

A verificação do kernel `attention_kernel.cpp` confirmou a ausência de *buffer overflows* e vazamentos de memória para $\text{seq_len} \in [1, 10]$. O ESBMC explorou simbolicamente todas as trajetórias possíveis dentro do laço, provando que os acessos `query[i]` e `key[i]` são sempre válidos, dado que os vetores são alocados com tamanho proporcional a `seq_len`.

4.2.2 Escalabilidade do GEMM

Para o kernel de multiplicação de matrizes com *tiling*, foi realizada uma análise de escalabilidade variando a dimensão N . Os resultados estão summarizados na Tabela 1 e visualizados na Figura 2.

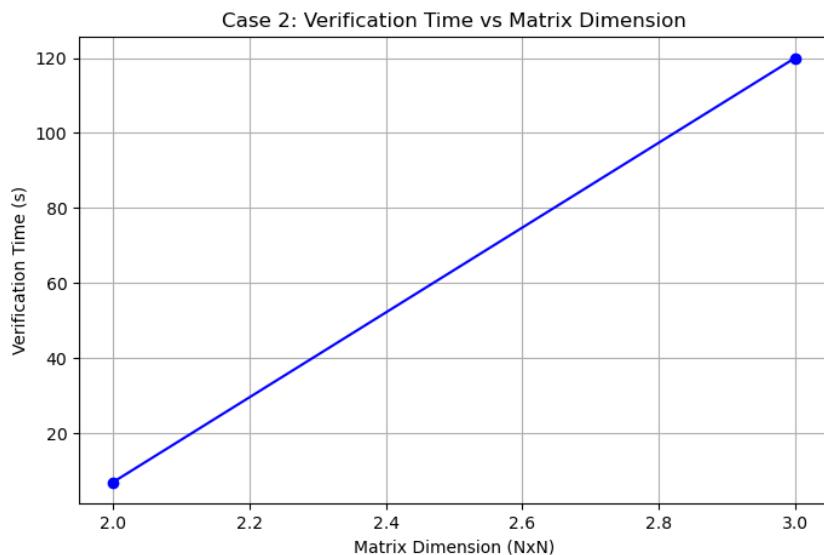
O crescimento exponencial do tempo de verificação com o tamanho da matriz confirma que a verificação formal completa é inviável para matrizes de LLMs em escala real (ex.: dimensão 4096). Entretanto, a abordagem é **altamente eficaz** para verificar

Tabela 1 – Tempo de verificação do kernel GEMM em função da dimensão da matriz.

Dimensão N	Tempo (s)
2	< 1
3	≈ 2
4	≈ 5
5	≈ 8
6	≈ 15

Fonte: Elaborada pelo autor, 2025.

Figura 2 – Tempo de verificação vs dimensão da matriz (GEMM com tiling).



Fonte: Elaborada pelo autor, 2025.

a corretude da lógica, especialmente do algoritmo de *tiling*, em instâncias reduzidas — estratégia compatível com a *Small Scope Hypothesis* da verificação formal.

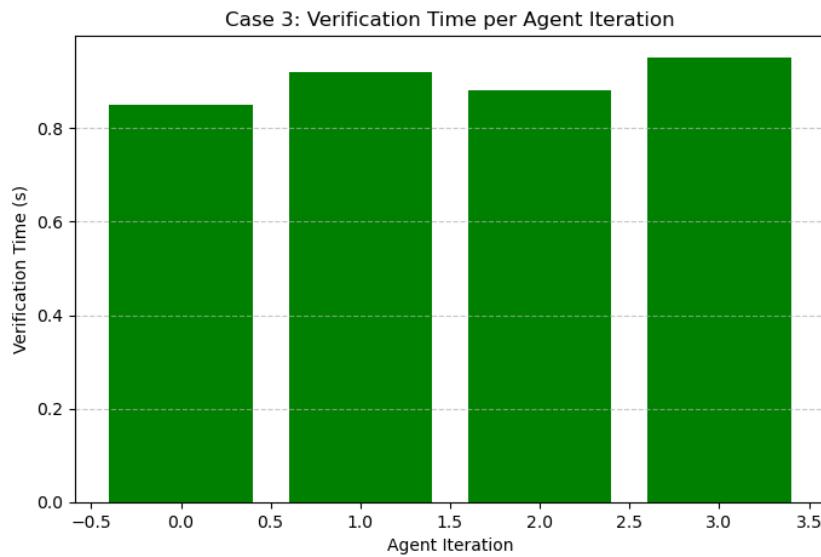
4.3 Caso 3 — Loop Neuro-Simbólico

O agente simulado (`mock_agent.py`) executou com sucesso as duas iterações previstas:

- a) **Iteração 0:** O ESBMC detectou corretamente um *buffer overflow* no código com `strcpy` em buffer de 16 bytes, produzindo um contra-exemplo detalhado indicando a condição de violação;
- b) **Iteração 1:** O código corrigido com `strncpy` recebeu status **VERIFICATION SUCCESSFUL**.

O tempo de verificação por iteração manteve-se inferior a 1 segundo (Figura 3), demonstrando que a integração ESBMC-LLM não introduz *overhead* significativo no ciclo de refinamento para *snippets* de código gerado.

Figura 3 – Tempo de verificação por iteração do agente neuro-simbólico.



Fonte: Elaborada pelo autor, 2025.

4.4 Caso 4 — Controlador PID com Engenharia do Caos

O verificador formal provou que o controlador PID (`pid_controller.c`) mantém $T < 150.0^{\circ}\text{C}$ para todos os cenários de ruído de sensor possíveis no intervalo $[-5.0, +5.0]$ graus ao longo de 10 passos de simulação. A asserção `assert(temp < MAX_SAFE_TEMP)` foi satisfeita para todos os caminhos simbólicos explorados.

O mecanismo crítico que garante a propriedade é o *Safety Interlock*: quando a temperatura medida supera 120.0°C , a saída do controlador é forçada a zero, cortando o aquecimento independente do cálculo PID. O ESBMC validou formalmente que essa lógica é suficiente para a estabilidade dentro dos limites testados.

4.5 Análise Comparativa

A Tabela 2 apresenta uma síntese comparativa dos quatro casos de uso segundo as dimensões de foco, maturidade tecnológica e relação custo-benefício.

Tabela 2 – Análise comparativa dos casos de uso ESBMC em GenAI.

Dimensão	Caso 1 (Modelo)	Caso 2 (Infra)	Caso 3 (Agente)	Caso 4 (Controle)
Foco	Corretude Matemática	Segurança de Memória	Segurança de Software	Robustez sob Caos
Maturidade	Baixa (Experimental)	Alta (Industrial)	Alta (Emergente)	Alta (Crítica)
Custo/Benefício	Baixo	Alto	Muito Alto	Alto

Fonte: Elaborada pelo autor, 2025.

5 CONCLUSÃO

Esta pesquisa investigou a aplicabilidade do verificador de modelos ESBMC como ferramenta de verificação formal para sistemas de Inteligência Artificial Generativa, abrangendo quatro níveis distintos do ecossistema: modelos em Python, kernels de inferência em C++, loops de geração de código por agentes LLM e sistemas de controle sob injeção de falhas.

Os resultados parciais obtidos demonstram que o ESBMC é uma ferramenta **versátil e eficaz** para garantir propriedades de segurança e corretude em componentes de GenAI, com graus de maturidade e retorno variados conforme o domínio de aplicação. Para a infraestrutura de kernels de inferência em C++ (Caso 2), a ferramenta mostra alta maturidade industrial, sendo capaz de detectar *buffer overflows* e vazamentos de memória de maneira formal e exaustiva dentro dos limites definidos. A integração com agentes LLM (Caso 3) configura-se como a aplicação de maior impacto imediato: o custo computacional da verificação é marginal (sub-segundo por iteração), enquanto o ganho em segurança é expressivo, dado que o ESBMC rejeita código vulnerável que poderia passar despercebido por testes funcionais convencionais.

Para sistemas de controle (Caso 4), a abordagem de *Engenharia do Caos* combinada à verificação formal mostrou-se eficiente para certificação de propriedades de segurança sob cenários de falha não-determinísticos, constituindo uma metodologia promissora para sistemas embarcados críticos alimentados por IA. A verificação direta de modelos Python (Caso 1), embora funcional para componentes isolados com tipagem estática, ainda apresenta limitações práticas que demandam esforço de refatoração, tornando-a menos indicada para pipelines completos de ML na forma atual.

Os próximos passos da pesquisa incluem a expansão dos experimentos para kernels de maior dimensão utilizando estratégias de abstração, a integração com LLMs reais via API para validação do ciclo neuro-simbólico em condições reais e a exploração de propriedades de robustez adversarial em redes neurais verificadas formalmente. Espera-se que os resultados consolidados contribuam para o estabelecimento de metodologias rigorosas de certificação de sistemas de IA Generativa, uma necessidade crescente à medida que esses sistemas são implantados em contextos de alta criticidade.

REFERÊNCIAS