Verificação Formal de Código Fonte de Sistema Embarcado em um Veículo Aéreo Não Tripulado

Ana Paula Fleck*, Guilherme Prudente da Silva*†, Leandro Buss Becker*
*Automation and Control Systems Engineering Dept
Federal University of Santa Catarina - Florianópolis, SC, Brazil
Emails: anakfleck@gmail, gprudente@gmail.com, leandro.becker@ufsc.br
† Eletrobrás - Rio de Janeiro, RJ, Brazil

***Abstract*—Este trabalho apresenta a aplicação de verificação formal em sistemas embarcados críticos, com foco na análise estática do código-fonte utilizando a ferramenta ESBMC. Com foco na detecção de vulnerabilidades em tempo de desenvolvimento, foram analisados 27 arquivos do código-fonte, totalizando mais de 10 mil linhas. Para contornar limitações da ferramenta, foram criados arquivos auxiliares contendo funções de entrada específicas, além da utilização de stubs para simular bibliotecas externas como lwIP e FreeRTOS. A análise permitiu identificar falhas relevantes, como acessos fora dos limites de arrays, uso indevido de ponteiros e vazamentos de memória. Os resultados demonstram o potencial da verificação formal para aumentar a robustez e a confiabilidade do sistema, ao mesmo tempo em que evidenciam os desafios relacionados à verificação modular e ao esforço manual necessário para estruturar os testes.**

***Index Terms*—Sistemas Embarcados, Sistemas Críticos de Segurança, Verificação Formal, Bounded Model Checking**

I. Introdução

Sistemas Críticos de Segurança (SCS) são aqueles cuja falha pode resultar em perda de vidas ou danos graves, exigindo um alto grau de confiabilidade [1]. O projeto ProVANT — uma colaboração entre UFMG, UFSC e Uni-Sevilha — desenvolve um SCS, que é Veículo Aéreo Não Tripulado (VANT) do tipo VTOL-CP destinado a missões de busca e salvamento [2], [3].

A verificação formal tem se destacado como uma abordagem rigorosa e sistemática para identificar falhas em SCS. Uma ferramenta de destaque é o ESBMC (*Efficient SMT-Based Bounded Model Checker*) [4], que verifica propriedades em programas C/C++ por meio da técnica de Bounded Model Checking (BMC) [5]. Caso uma violação seja detectada, a ferramenta gera um contra-exemplo que ilustra a sequência de eventos que leva à falha, facilitando sua compreensão e correção, e contribuindo para o aumento da confiabilidade do sistema.

Este trabalho apresenta a aplicação do ESBMC à análise formal do sistema embarcado de controle do ProVANT, composto por 10.674 linhas de código-fonte, 220 funções e 27 arquivos. O objetivo do trabalho é identificar vulnerabilidades no código, como por exemplo falhas de alocação de memória, uso de ponteiros inválidos, estouros de buffer, entre outras propriedades críticas, permitindo intervenções corretivas com base em evidências formais.

O restante do artigo está organizado da seguinte forma. A Seção II descreve o ESBMC e sua base, o Bounded Model Checking. Na Seção III é apresentada a abordagem proposta. Os resultados obtidos são apresentados na Seção IV. Por fim, nossas conclusões são apresentadas na Seção V.

II. Background Teórico

A. Bounded Model Checking

Model Checking é uma técnica de verificação formal que explora sistematicamente todos os estados possíveis de um sistema, com o objetivo de verificar se determinadas propriedades são satisfeitas ao longo de sua execução [6], indo além do possível com testes tradicionais. Quando uma propriedade é violada, o verificador fornece um contraexemplo com uma sequência de estados que leva à falha, auxiliando na sua depuração e correção.

Proposto originalmente por Biere et al. [5], o Bounded Model Checking (BMC) restringe a verificação a um número finito de passos de execução, definidos por um limite *k*. Em vez de explorar todos os estados possíveis, o BMC busca contraexemplos com até *k* transições, tornando o processo mais eficiente, especialmente na detecção de erros em fases iniciais.

Para isso, o BMC traduz o comportamento do sistema e a negação da propriedade a ser verificada em uma fórmula lógica, geralmente expressa em termos de satisfatibilidade booleana (SAT) ou satisfatibilidade módulo teorias (SMT). Essa fórmula é avaliada por um solver, que verifica se existe uma atribuição que viola a propriedade dentro do limite de *k* passos. Caso exista, um contraexemplo é extraído e apresentado ao usuário. A fórmula do BMC pode ser representada da seguinte forma:

$$\text{BMC}_{\varphi}(k) = I(s_1) \wedge \bigwedge_{i=1}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=1}^{k} \neg\varphi(s_i) \quad (1)$$

Aqui, $I(s_1)$ representa o conjunto de estados iniciais. A conjunção $\bigwedge_{i=1}^{k-1} T(s_i, s_{i+1})$ descreve as transições ao longo de k passos. Já a disjunção $\bigvee_{i=1}^{k} \neg\varphi(s_i)$ indica a negação da propriedade φ em algum estado. Esses componentes formam um problema factível caso exista um contraexemplo de comprimento k ou menor, indicando a violação da propriedade no limite analisado [7].

B. ESBMC

O ESBMC [4] é uma ferramenta open-source para verificação formal de software baseada na técnica de BMC e

no uso de SMT solvers. São suportados sistemas nas linguagens C/C++, Solidity, Kotlin, Python e Java. Sua arquitetura é composta por três componentes principais: o gerador de grafo de controle de fluxo (CFG), responsável pela análise sintática e criação do programa intermediário em GOTO; o motor de execução simbólica, que transforma o programa em uma representação SSA (Static Single Assignment), realiza o desenrolar de laços e gera fórmulas para verificação; e o backend SMT, que converte essas fórmulas em instâncias resolvíveis por solvers como Z3 e Boolector.

Durante a verificação, o ESBMC analisa automaticamente propriedades de segurança e corretude, como acesso fora dos limites de arrays, uso inválido de ponteiros, estouros aritméticos, ocorrência de *NaN* (*Not a Number*) em operações de ponto flutuante, divisão por zero, vazamentos de memória, conflitos de concorrência, entre outras. A verificação é feita pela linha de comando, com parâmetros configuráveis, como o arquivo testado, o tipo de propriedade, o solver e opções específicas. Um exemplo seria: `esbmc arquivo.c -nan-check -incremental-bmc`, onde `arquivo.c` é o código-fonte, `-nan-check` define a propriedade e `-incremental-bmc` o solver usado. Caso alguma propriedade seja violada, o ESBMC retorna uma sequência de estados com os valores das variáveis que levaram à falha. Esse contraexemplo ajuda o desenvolvedor a rastrear a origem do erro e identificar a vulnerabilidade.

São suportados dois métodos de verificação avançados: o *k-induction*, que utiliza base, condição de avanço e passo indutivo para provar a correção do sistema em múltiplas iterações de laço; e o *BMC incremental*, que verifica o código progressivamente ao aumentar o limite de desenrolamento dos laços. Ambas as abordagens visam equilibrar eficiência e profundidade de verificação, sendo adequadas para sistemas complexos e com comportamento iterativo. Essas características tornam o ESBMC especialmente adequado para SCS, como tratado nesse trabalho.

III. Abordagem Proposta

A primeira parte da seção apresenta um panorama geral do sistema de controle embarcado do VTOL-CP em desenvolvimento no ProVANT. Em seguida, descreve-se a metodologia aplicada, com foco na estruturação da análise formal, no processo de preparação dos arquivos e na execução dos testes.

A. VTOL-CP do ProVANT

O VTOL-CP conta com uma plataforma embarcada responsável pelo processamento a bordo, sensores e atuadores para controle de voo. O sistema é dividido em duas camadas principais: o High Level Hardware (HLH), composto por um Jetson TX2 e um Raspberry Pi 4, e o Low Level Hardware (LLH), formado por duas placas STM32 Nucleo-F767ZI redundantes, responsáveis pela interface com sensores e atuadores [8], aumentando, assim, a confiabilidade do sistema.

O código analisado neste trabalho, no contexto da verificação formal, corresponde especificamente à parte embarcada executada nas placas STM32 Nucleo-F767ZI, responsáveis pela camada de baixo nível (LLH) do sistema. Esse código, escrito em linguagem C, é responsável por diversas funcionalidades essenciais, incluindo drivers de sensores e atuadores, e módulos de comunicação (como protocolo de mensagens e interface Ethernet). Também fazem parte da base de código os arquivos de integração do sistema, sendo um dedicado à inicialização da arquitetura do VANT, e três que implementam as principais *threads* de execução, responsáveis respectivamente, pela comunicação, processamento de dados e controle. Esses componentes representam a base crítica do funcionamento do VANT e, por isso, foram o foco da aplicação das técnicas de verificação formal descritas neste artigo.

B. Metodologia de Verificação Formal

A metodologia adotada neste trabalho consistiu em aplicar o verificador ESBMC ao sistema embarcado do ProVANT para identificar falhas e aumentar sua robustez e confiabilidade. Primeiro, foi feita uma análise detalhada do código do projeto para estruturar a verificação de forma incremental. A abordagem adotada priorizou, inicialmente, os arquivos mais simples do sistema — com funções pontuais, de menor complexidade e sem dependências externas — por serem mais diretos de analisar e facilitarem a familiarização com o comportamento do verificador. Depois, foram verificados arquivos de complexidade intermediária, com maior extensão, várias funções e certo grau de interdependência. Por fim, a análise focou nos arquivos mais complexos e centrais do sistema, responsáveis por funções críticas e pela integração dos componentes da arquitetura embarcada.

Cada arquivo foi testado individualmente. Para isso, criou-se uma cópia contendo todas as funções do módulo, com as implementações reorganizadas de forma coerente. Nos casos de forte acoplamento com bibliotecas como lwIP, UDP e FreeRTOS, foram utilizados *stubs* — versões simplificadas de funções ou componentes que substituem temporariamente as dependências reais. Esses stubs simulam o comportamento esperado dessas dependências, permitindo a execução dos testes de forma independente e controlada. Embora tenha sido feita uma tentativa de automatizar a criação das funções `main`, optou-se, ao final, pela construção manual dessas funções para cada arquivo analisado, a fim de garantir que todas as funcionalidades relevantes fossem exercitadas. Esse processo demandou a implementação manual de chamadas a mais de 220 funções em 27 arquivos, representando esforço e tempo significativos apenas na preparação do ambiente de análise.

As Listagens 1 e 2 exemplificam o uso do ESBMC. A Listagem 1 representa uma simulação da leitura de um sensor. Já a Listagem 2 mostra uma função `main` criada para executar essa função durante a verificação, fornecendo entradas não determinísticas e garantindo que o código analisado seja efetivamente exercitado pelo verificador.

```
1 int read_sensor_data(int sensor_id, float *value) {
2     *value = 42.0; // Simula leitura de sensor
3     return 0; }
```

Listing 1. Exemplo de código-fonte

```
1 int main() {
2     int sensor_id = nondet_int();
3     float val;
4     read_sensor_data(sensor_id, &val);
5     return 0; }
```

Listing 2. Exemplo de função main criada para verificação formal

A execução dos testes com o ESBMC foi parcialmente automatizada por meio de um script, responsável por aplicar combinações sistemáticas de parâmetros, propriedades e solvers. Com isso, foi possível ampliar a cobertura da análise, padronizar as execuções e otimizar o tempo necessário para verificação. Quando alguma propriedade era violada, o ESBMC reportava o erro diretamente no terminal, permitindo a análise e classificação manual dos resultados obtidos.

IV. Resultados

Os resultados foram obtidos com a aplicação do verificador ESBMC em 27 arquivos do sistema embarcado do ProVANT. Até o momento da finalização deste trabalho, 13 foram validados sem vulnerabilidades, 8 apresentaram falhas e 6 não puderam ser totalmente testados por limitações de tempo e escopo. A Tabela I apresenta um resumo dos resultados obtidos. Em seguida, cada falha identificada será descrita em mais detalhes.

TABLE I
Resumo dos Resultados

Módulo	Status	Falha Identificada
`c_io_sonar.c`	Corrigido	Acesso array fora dos limites
`typedefs.c`	Corrigido	Vazamento de memória
`pv_type_threadData.c`	Ignorado	Vazamento de memória e ponteiro nulo
`c_io_EPOS.c`	Corrigido	Possível divisão por zero (`NaN`)
`c_CSC.c`	Ajustado	Erro na lógica de programação
`VANT4_Architecture.c`	Ajustado	Erro na lógica de programação

A. Acessos fora dos limites de arrays

No arquivo `c_io_sonar.c`, responsável pela comunicação com o sonar, foi identificado um acesso fora dos limites de um array. A função `sonar_receive_i()`, mostrada na Listagem 3, percorria o vetor `rx_sonar` até encontrar o caractere de início de mensagem. Porém, a condição de parada `start == SONAR_BUF_SZ + 1` (linha 8) permitia acesso ao índice `SONAR_BUF_SZ`, fora do intervalo válido (0 a `SONAR_BUF_SZ - 1`), podendo causar comportamento indefinido. A correção, na Listagem 4, ajustou a verificação para `start == SONAR_BUF_SZ`, garantindo a segurança da operação. A solução foi aplicada, testada no ESBMC e validada com sucesso.

```
1  int sonar_receive_i () {
2      int start = 0;
3      while (1) {
4          if (rx_sonar[start] == 'R')
5              break;
6
7          start++;
8          if (start == SONAR_BUF_SZ + 1)
9              return -1;
10     }   ... }
```

Listing 3. Função com acesso fora dos limites

```
1  int sonar_receive_i () {
2      int start = 0;
3      while (1) {
4          if (rx_sonar[start] == 'R')
5              break;
6          start++;
7          if (start == SONAR_BUF_SZ)
8              return -1;
9      }  ...  }
```

Listing 4. Versão corrigida da função

B. Vazamento de memória e uso indevido de ponteiros

No módulo `typedefs.c`, foi detectado um vazamento de memória causado pela ausência de liberação após uma falha na alocação dinâmica. A função afetada, apresentada na Listagem 5 alocava um espaço de memória e, em caso de falha (por exemplo, por insuficiência de memória), retornava sem liberar a memória previamente alocada. A correção do problema, apresentada na Listagem 6, consistiu na inclusão de uma chamada à função `free()` na linha 6, conforme sugerido pelas melhores práticas de gerenciamento de memória e, com isso, tornando o programa resiliente.

```
1  int c_type_radioControl_initRadio(pv_type_radioControl *
       radio, u8 id, int channel_num){
2      ...
3      i8* auxBuf_channels = malloc(auxBufSz_channels);
4      if (auxBuf_channels == NULL) {
5          return -1;
6      }  ...  }
```

Listing 5. Função com vazamento de memória

```
1  int c_type_radioControl_initRadio(pv_type_radioControl *
       radio, u8 id, int channel_num){
2      ...
3      i8* auxBuf_channels = malloc(auxBufSz_channels);
4      if (auxBuf_channels == NULL) {
5          free(auxBuf.raw);
6          return -1;
7      } ... }
```

Listing 6. Versão corrigida da função

Já no arquivo `pv_type_threadData.c`, foram encontrados problemas relacionados a ponteiros nulos e possíveis vazamentos de memória. No entanto, após consulta com os desenvolvedores, foi esclarecido que esse módulo se tratava de uma implementação incompleta e atualmente não é mais utilizado no sistema. Assim, os erros foram registrados, mas desconsiderados para fins de impacto na verificação formal.

C. Divisão por zero e ocorrências de NaN

Na análise do módulo `c_io_EPOS.c`, foi detectada uma operação de divisão em ponto flutuante que poderia resultar em um valor *NaN*. A função em questão realiza a conversão da posição lida de um servomotor, expressa em incrementos

do encoder, para um valor em graus, utilizando a razão entre o número de incrementos e o total de incrementos por volta do servo, multiplicada por 360.

Essa conversão é sensível ao valor do denominador, pois se o número total de incrementos por volta estiver zerado, a divisão resultará em *NaN*, afetando cálculos posteriores e a integridade do sistema. A Listagem 7 apresenta um trecho representativo do código onde esse risco ocorre.

Para evitar a propagação de valores indefinidos no sistema, foi adicionada uma verificação condicional antes da divisão, garantindo que o denominador não seja zero. A Listagem 8 apresenta essa modificação, que garante que a operação seja realizada apenas em condições válidas, prevenindo resultados inesperados durante a execução e contribuindo para a robustez do sistema.

```
1 static float servoIncToDeg(pv_type_EPOS_Servo *me, int
      pos_inc){
2   float pos_deg = (float)pos_inc / (float)me->
        encoderTurns * 360.0;
3   if(pos_deg > 360.0) pos_deg -= 360.0;
4   if(pos_deg <  -360.0) pos_deg += 360.0;
5   return pos_deg; }
```

Listing 7. Função com risco de *NaN* por divisão por zero

```
1 if (me->encoderTurns < 0.000001 && me->encoderTurns >
      -0.000001) {
2     return EPOS_ERR_INVALID;   }
```

Listing 8. Verificação para evitar *NaN*

D. Contribuições adicionais da análise formal

A utilização do ESBMC revelou não apenas vulnerabilidades diretamente relacionadas a propriedades de segurança, mas também comportamentos inesperados da ferramenta e problemas gerais de codificação. Em diferentes módulos — como `c_CSC.c`, `VANT4_Architecture.c` e `c_comm_frame.c` — o ESBMC interpretou incorretamente chamadas à função `realloc()`. Pela especificação em C, o uso de `realloc(NULL, size)` deve ser equivalente a um `malloc()`; no entanto, o ESBMC interpretou incorretamente esse uso como uma violação de ponteiro inválido. Essa inconsistência foi testada isoladamente, documentada e reportada à equipe de desenvolvimento do ESBMC. Ajustes no código contornaram o problema, e os testes posteriores foram bem-sucedidos.

A verificação também auxiliou na detecção de outros problemas de codificação, como erros de digitação, funções incompletas e más práticas de alocação de memória. Em um dos módulos analisados, por exemplo, foi identificado um comando condicional (`if`) sem corpo associado; ou seja, a condição era avaliada, mas nenhuma ação era tomada caso fosse verdadeira. Como era uma verificação após uma tentativa de alocação de memória, o código funcionava em situações ideais, mas permanecia vulnerável em caso de falha na alocação. A correção consistiu em inserir o comando `return -1`, garantindo o tratamento adequado do erro.

V. Conclusões e Próximas Etapas

Este trabalho apresentou a aplicação da ferramenta ESBMC para a verificação formal do código-fonte do sistema embarcado de controle do projeto ProVANT. A abordagem adotada consistiu em realizar uma análise sistemática dos arquivos do projeto, organizando-os conforme sua complexidade e grau de dependência. Ao todo foram analisados 27 arquivos fonte do sistema, totalizando 10.674 linhas de código. Como resultado, 13 arquivos foram validados sem apresentar vulnerabilidades, 8 apresentaram falhas e 6 não puderam ser testados integralmente devido à complexidade envolvida.

Entre as principais vulnerabilidades encontradas, destacam-se um acesso fora dos limites do array, uso incorreto na realocação de memória, além de falhas na verificação de ponteiros nulos. A análise também revelou trechos de código obsoletos ou não utilizados, evidenciando a importância da verificação formal não apenas para encontrar falhas, mas também para promover uma revisão estrutural do projeto.

Como próxima etapa, pretende-se ampliar a análise para os arquivos restantes e para novas implementações do projeto atualmente em desenvolvimento. Além disso, será investigada a integração entre módulos, uma vez que essas interações podem revelar vulnerabilidades não detectadas nos testes realizados de forma isolada.

Também está previsto o aprimoramento do processo de automação da preparação dos arquivos e da execução dos testes, com o objetivo de torná-lo mais robusto, reduzir o esforço manual envolvido e aumentar a escalabilidade da verificação. Idealmente, busca-se desenvolver uma solução flexível e reutilizável, que possa ser aplicada em outros projetos além do projeto ProVANT, agregando valor ao processo de verificação formal como um todo.

Agradecimentos

O projeto PROVANT conta com financiamento do edital universal 2023 do CNPq. A co-autora Ana foi bolsista de iniciação científica PIBIC (2024-2025) do CNPq.

Referencias

[1] J. C. Knight, "Safety critical systems: challenges and directions," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 547–550. [Online]. Available: https://doi.org/10.1145/581339.581406

[2] A. V. Lara, "Design of an embedded system architecture for a SCS," Master's thesis, 2019. [Online]. Available: http://hdl.handle.net/1843/76458

[3] F. S. Gonçalves, "Integrated method for designing complex cyber-physical systems," Ph.D. dissertation, Universidade Federal de Santa Catarina, Florianópolis, Brazil, 2018.

[4] L. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 331–340. [Online]. Available: https://doi.org/10.1145/1985793.1985839

[5] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Tools and Algorithms for the Construction and Analysis of Systems*, W. R. Cleaveland, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207.

[6] C. Baier and J.-P. Katoen, "Principles of model checking," p. 994, 2008.

[7] J. O. de Souza, "Lsverifier: a bmc approach to identify security vulnerabilities in c open-source software projects," Master's thesis, 2023, faculdade de Tecnologia. [Online]. Available: https://tede.ufam.edu.br/handle/tede/10010

[8] R. R. Ferreira *et al.*, "Runtime verification of low-level software in a vtol uav," 2025.