UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

MÁRIO DA SILVA ARAÚJO

ANÁLISE DE SEGURANÇA DA ARQUITETURA RISC-V DE CÓDIGO ABERTO

FORTALEZA
2025

MÁRIO DA SILVA ARAÚJO

ANÁLISE DE SEGURANÇA DA ARQUITETURA RISC-V DE CÓDIGO ABERTO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Elétrica do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia Elétrica.

Orientador: Prof. Dr. Paulo Peixoto Praça

FORTALEZA

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A69a Araújo, Mário da Silva.
Análise de segurança da arquitetura RISC-V de código aberto / Mário da Silva Araújo. – 2025.
60 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Tecnologia, Curso de Engenharia Elétrica, Fortaleza, 2025.
Orientação: Prof. Dr. Paulo Peixoto Praça.

1. Processador. 2. Análise Formal. 3. Injeções de Falhas. 4. Segurança de Processadores. I. Título.
CDD 621.3

MÁRIO DA SILVA ARAÚJO

ANÁLISE DE SEGURANÇA DA ARQUITETURA RISC-V DE CÓDIGO ABERTO

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Elétrica do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia Elétrica.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Paulo Peixoto Praça (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Luiz Henrique Silva Colado Barreto
Universidade Federal do Ceará (UFC)

Msc. Paulo Honorio Filho
Universidade Federal do Ceará (UFC)

Dedico este trabalho aos meus pais, que sempre acreditaram em mim e me apoiaram em cada passo da minha jornada. Sem o amor e a confiança de vocês, nada disso seria possível.

AGRADECIMENTOS

Gostaria de expressar minha profunda gratidão a todos os membros do Departamento de Engenharia Elétrica da Universidade Federal do Ceará (UFC), por oferecerem um curso de qualidade que contribuiu significativamente para minha formação acadêmica e profissional.

Agradeço aos meus amigos da faculdade, que estiveram ao meu lado durante toda a minha jornada acadêmica, proporcionando apoio, companheirismo e momentos de aprendizado. Em especial, agradeço a Douglas Militão, Lívia Belo, José Eduardo Holanda, Jéssica Feitosa, Michelly Karen Diógenes, Isadora Horita, João Victor Andrade, Dionatan Argenta, Iago Aguiar, Guilherme Teixeira, Carlos Humberto, Bruno Marvin e Leonardo Alves, por toda a ajuda e amizade ao longo dos anos. Também sou muito grato ao Laboratório de Energias Alternativas (LEA) e ao Programa de Educação Tutorial (PET) da UFC, que me auxiliaram tanto nas disciplinas quanto em atividades externas, enriquecendo minha experiência acadêmica.

Minha sincera gratidão ao professor doutor Paulo Cesar Marques de Carvalho, por me aceitar no Laboratório LEA e me proporcionar a oportunidade de participar do projeto de Iniciação Científica (PIBIC), o que possibilitou a publicação do artigo sobre previsão de energia solar gerada por painéis solares por meio do uso de inteligência artificial. Foi uma experiência única e de grande aprendizado.

Agradeço, também, aos professores doutores João Baptista Martins (Universidade Federal de Santa Maria - UFSM) e Jarbas Aryel Nunes da Silveira (Universidade Federal do Ceará - UFC), pela concessão da bolsa de estudos Brasil France Ingénieur Technologie (BRAFITEC), que me possibilitou estudar na École des Mines de Saint-Étienne (EMSE) e obter o duplo diploma, com apoio da CAPES. Essa oportunidade foi fundamental para o meu crescimento acadêmico e profissional.

Por fim, quero dedicar um agradecimento especial aos meus familiares, em particular aos meus pais, Mário Marques de Araújo e Eranilda da Silva Araújo, que sempre me apoiaram, acreditaram em mim e estiveram presentes em cada etapa dessa caminhada.

“As pessoas costumam dizer que a motivação não dura sempre. Bem, nem o efeito do banho, por isso recomenda-se diariamente.”

(Zig Ziglar)

RESUMO

Neste trabalho, é apresentada uma análise de segurança aplicada ao processador de código aberto CV32E40S, fornecido pela fundação OpenHWGroup. O estudo foi conduzido durante estágio técnico na empresa *Commissariat de l'Énergie Atomique*, com foco no modelo de atacante e no uso de injeção de falhas como técnica principal. Para isso, a análise de segurança foi realizada através da ferramenta μArchiFI que foi desenvolvido pela equipe de segurança de hardware a fim de efetuar uma análise formal de um sistema completo, englobando tanto o lado hardware quanto o lado software de um sistema embarcado, nos quais os componentes estão submetidos às injeções de falhas.

O processador CV32E40S é um núcleo RISC-V 32 bits com um pipeline de 4 etapas (IF, ID, EXE e WB), concedido para aplicações de segurança. Ele utiliza uma extensão personalizada nomeada de *Xsecure* que inclui funcionalidades de segurança tais que o reforçamento do contador de programa e a verificação de paridade durante os acessos à memória. O objetivo dessa análise é de formalizar as contramedidas implementadas no processador CV32E40S.

A avaliação leva em consideração o programa software executado pelo processador. Para este feito, a coleção de códigos públicos FISSC (*Fault Injection and Simulation Secure Collection*) foi utilizada para avaliar a robustez do sistema face às injeções de falhas.

Palavras-chave: Processador. Análise Formal. Injeções de Falhas. Segurança de Processadores.

ABSTRACT

This work presents a security analysis applied to the open-source processor CV32E40S, provided by the *OpenHWGroup* foundation. The study was conducted during a technical internship at the *Commissariat de l'Énergie Atomique*, focusing on the attacker model and using fault injection as the main technique. For this purpose, the security analysis was conducted using the μArchiFI tool, developed by the hardware security team to perform a formal analysis of a complete system, encompassing both the hardware and software sides of an embedded system, where components are subjected to fault injections.

The CV32E40S processor is a 32-bit RISC-V core with a 4-stage pipeline (IF, ID, EXE, and MB) designed for security applications. It employs a custom extension named *Xsecure*, which includes security features such as program counter hardening and parity checking during memory accesses. The objective of this analysis is to formalize the countermeasures implemented in the CV32E40S processor.

The evaluation considers the software program executed by the processor. To this end, the public FISSC (*Fault Injection and Simulation Secure Collection*) code suite was used to assess the system's robustness against fault injections.

Keywords: Processor. Formal Analysis. Fault Injections. Processor Security.

LISTA DE FIGURAS

Figura 1 – Circuito somador sob injeção de falha no fio c_i. 14
Figura 2 – Arquitetura do *pipeline* do processador de 5 estágios. 15
Figura 3 – Arquitetura de *pipeline* de processador de 5 estágios corrompida. 16
Figura 4 – Exemplificação de como a corrupção de dados a nível de *hardware* é transmitida para o nível do *software*. 17
Figura 5 – Exemplo de como o ataque de injeção de falha de *hardware* pode contornar a verificação de senha. 18
Figura 6 – Código de verificação do Personal Identification Number (PIN) (Heydemann 2024). 18
Figura 7 – Arquitetura e rede de ferramentas de verificação μArchiFI. 23
Figura 8 – Emulação de falhas por meio de multiplexadores. 24
Figura 9 – Exemplo do uso da passagem FaultRTLIL. 24
Figura 10 – Esquemático do núcleo CV32E40P. 28
Figura 11 – Ambiente de programação para o CV32E40P. 29
Figura 12 – Fluxograma da verificação de PINs - função *verifyPIN* versão 7. 31
Figura 13 – Propriedades estabelecidas para o CV32E40P verificar. 34
Figura 14 – Comandos Yosys utilizados para injetar falhas no CV32E40P. 34
Figura 15 – Cronologia para a avaliação da robustez do CV32E40P + *verifypin_v7.c*. . . 35
Figura 16 – Resultado da verificação formal gerada pelo solucionador pono. 36
Figura 17 – Esquemático do núcleo CV32E40S. 37
Figura 18 – Fluxograma da inicialização do PIN com o comando para inicializar a contramedida desejada. 41
Figura 19 – Fluxogramas do processo de verificação de PINs da versão zero. 42
Figura 20 – Comandos Yosys utilizados para injetar falhas no CV32E40S. 43
Figura 21 – Propriedades estabelecidas para o CV32E40S verificar. 44

LISTA DE TABELAS

Tabela 1 – Endereços e valores de cada variável usada no código *verifypin_v7.c*. 33
Tabela 2 – Possibilidades e suas interpretações para as variáveis g_authenticated e g_countermeasure. 33
Tabela 3 – Intervalos para os parâmetros rnddummyfreq. 38
Tabela 4 – Endereços e valores de cada variável usada no código *verifypin_v0.c*. 43
Tabela 5 – Tempo de simulação para cada configuração executada no Questasim. . . . 44
Tabela 6 – Resultados da verificação formal. 45

LISTA DE ABREVIATURAS E SIGLAS

BMC	Bounded Model Checking
BSP	Board Support Package
CSR	Control and Status Register
ECC	Error Correction Code
EXE	Execution
FIA	Fault Injection Attack
FISSC	Fault Injection and Simulation Secure Collection
HDL	Hardware Description Language
ID	Instruction Decode
IF	Instruction Fetch
IoT	Internet of Things
ISA	Instruction Set Architecture
LFSR	Linear Feedback Shift Register
MEM	Memory
OBI	Open Bus Interface
PC	Program Counter
PIN	Personal Identification Number
PMA	Physical Memory Attribution
PMP	Physical Memory Protection
RISC-V	Reduced Instruction Set Computer - Five
RTL	Register Transfer Level
RTLIL	RTL Intermediate Language
SMT	Satisfiability Modulo Theories
UVM	Universal Verification Methodology
VCD	Value Change Dump
WB	Write Back

SUMÁRIO

1 INTRODUÇÃO 13

1.1 Contexto e problemática 13

1.2 Objetivos do trabalho 19

2 ESTADO DA ARTE 20

2.1 Ferramentas de *software* utilizados 20

2.2 Workflow uArchiFI 21

3 DESENVOLVIMENTO 27

3.1 Apresentação do CV32E40P 27

3.1.1 Geração de códigos binários 28

3.1.2 Simulação do microprocessador + programa binário 31

3.1.3 Conversão do hardware 32

3.1.4 Avaliaçao da robustez do CV32E40P 32

3.2 Desenvolvimento do ambiente de trabalho - CV32E40S 37

3.2.1 Preparação das contramedidas no código 41

3.2.2 Preparação do código a ser executado 42

3.2.3 Avaliação da robustez do CV32E40S 43

3.2.4 Resultados obtidos 44

4 CONCLUSÃO 46

4.1 Trabalhos futuros 46

4.2 Dificuldades encontradas 47

REFERÊNCIAS 49

APÊNDICES 51

APÊNDICE A – Códigos utilizados neste trabalho 51

1 INTRODUÇÃO

1.1 Contexto e problemática

Os ataques de injeção de falhas (do inglês: Fault Injection Attack (FIA)) consistem na perturbação do funcionamento normal dos sistemas incorporados para gerar erros. Estes erros podem ser propagados através da microarquitetura, afetando os cálculos do sistema. As consequências se manifestam ao nível do *software*, onde os resultados produzidos não correspondem às expectativas. Embora um ataque de injeção de falhas seja sempre considerado um ataque ao *hardware*, pode ter implicações no *software*. As falhas induzidas podem ser exploradas no *software*, criando novas vulnerabilidades que não estavam inicialmente presentes.

Os ataques de injeção de falhas podem ser caracterizados como:

- **Ataques de *hardware***: estes ataques requerem acesso físico ao dispositivo alvo. Utilizando ferramentas como sondas JTAG, analisadores lógicos, microscópios eletrônicos, etc., os atacantes podem extrair informações do consumo de energia de uma operação para deduzir o que o dispositivo está a executar. Além disso, os atacantes podem utilizar técnicas para modificar o funcionamento físico do *hardware* para provocar falhas, ou seja, operações não programadas para alterar o percurso inicialmente planejado.
- **Ataques ao *software***: exploram vulnerabilidades no código, como erros de programação ou falhas de segurança não corrigidas. Em ataques de injeção de falhas, a falha do *hardware* cria ou amplifica estas vulnerabilidades, abrindo a possibilidade de exploração do *software*. As falhas induzidas modificam a forma como o *software* funciona, permitindo ao atacante contornar os mecanismos de segurança ou obter acesso não autorizado.

Existem várias técnicas para produzir uma falha, como:

- ***Clock Glitching***: manipula o sinal de relógio para provocar erros na seção de temporização de instruções (*clock setup*);
- ***Voltage Glitching***: modifica temporariamente a tensão de alimentação do circuito para induzir erros no processamento de dados ou na execução de instruções do circuito;
- **Eletromagnético**: utiliza impulsos eletromagnéticos para interferir com o funcionamento do circuito eletrônico, provocando erros de cálculo;
- **Temperatura**: expõe o circuito a variações extremas de temperatura para induzir falhas;
- **Pulso de laser**: usa pulsos de laser para atingir áreas específicas do circuito, causando falhas nos transistores do sistema.

Como resultado, o efeito gerado é manifestado no circuito, modificando a operação correta dos componentes usados pelo microprocessador. O resultado final pode variar de acordo com a duração (transitório, permanente, destrutivo), o tipo (inversão, *set*, *reset*, determinístico, aleatório), a granularidade (*bit* único, *multibits*, *byte*, palavra) etc.

Por exemplo, a Figura 1 mostra o circuito para uma operação de adição completa, porém há uma falha no fio c_i. Portanto, se a operação for realizada com os valores $a_i = b_i = c_i = 0$, o resultado esperado deverá ser $s_o = c_o = 0$. Entretanto, devido à falha, o valor de c_i será modificado. Portanto, se o efeito da modificação for definir o estado para um nível baixo, ou seja, 0, o resultado final não será modificado; caso contrário, o estado será definido para um nível alto, ou seja, 1, e o resultado final será modificado para $s_o = 1$ e $c_o = 0$.

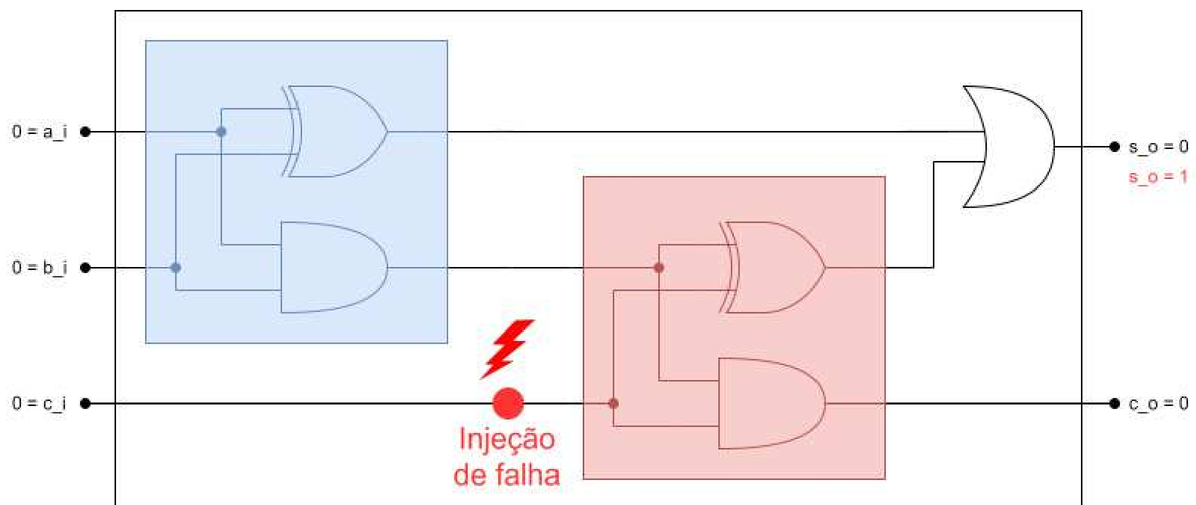


Figura 1 – Circuito somador sob injeção de falha no fio c_i.

A microarquitetura de um microprocessador geralmente contém um *pipeline* composto de vários estágios, geralmente contendo 5 estágios (como mostrado na Figura 2), embora isso possa variar dependendo do projeto específico. Cada estágio realiza parte da operação necessária para executar uma instrução. Os principais estágios são:

- **Estágio de busca de instruções - Instruction Fetch (IF)**: esse estágio carrega as instruções a serem executadas da memória de instruções. Ele usa um Program Counter (PC) que é incrementado para apontar para a próxima instrução;
- **Estágio de decodificação de instruções - Instruction Decode (ID)**: depois que o estágio IF carrega uma instrução, ela é transmitida ao estágio ID, que é responsável por decodificá-la e ler os operandos dos registros;
- **Estágio de execução - Execution (EXE)**: esse estágio recebe a instrução decodificada e a

executa. As execuções podem ser: operações aritméticas, operações lógicas, cálculos de endereço, resoluções de ramificação;

- **Estágio Memory (MEM)**: esse estágio acessa a memória de dados para executar operações de leitura ou gravação para instruções de carregamento ou armazenamento;
- **Estágio Write Back (WB)**: esse estágio grava os resultados da operação no banco de registros.

Em um *pipeline*, várias instruções são executadas simultaneamente em diferentes estágios, o que pode levar a riscos (*Hazards*). Há três tipos de riscos:

- **Risco de dados**: quando uma instrução depende do resultado de uma instrução anterior que não foi concluída;
- **Risco de controle**: quando o *pipeline* precisa tomar uma decisão com base no resultado de uma instrução de ramificação que não foi resolvida;
- **Risco estrutural**: quando dois estágios diferentes do *pipeline* estão competindo pela mesma fonte de *hardware*.

Técnicas como o *forwarding* podem ser usadas para resolver determinados riscos de dados, redirecionando diretamente os resultados das instruções para os próximos estágios do *pipeline* sem esperar que o ciclo de gravação seja concluído. No entanto, nem todos os processadores implementam o *forwarding* e, nos casos em que essa técnica não está disponível, são introduzidas paradas (*stalls*) no *pipeline* para atrasar determinadas instruções até que as dependências sejam resolvidas e para evitar erros de execução.

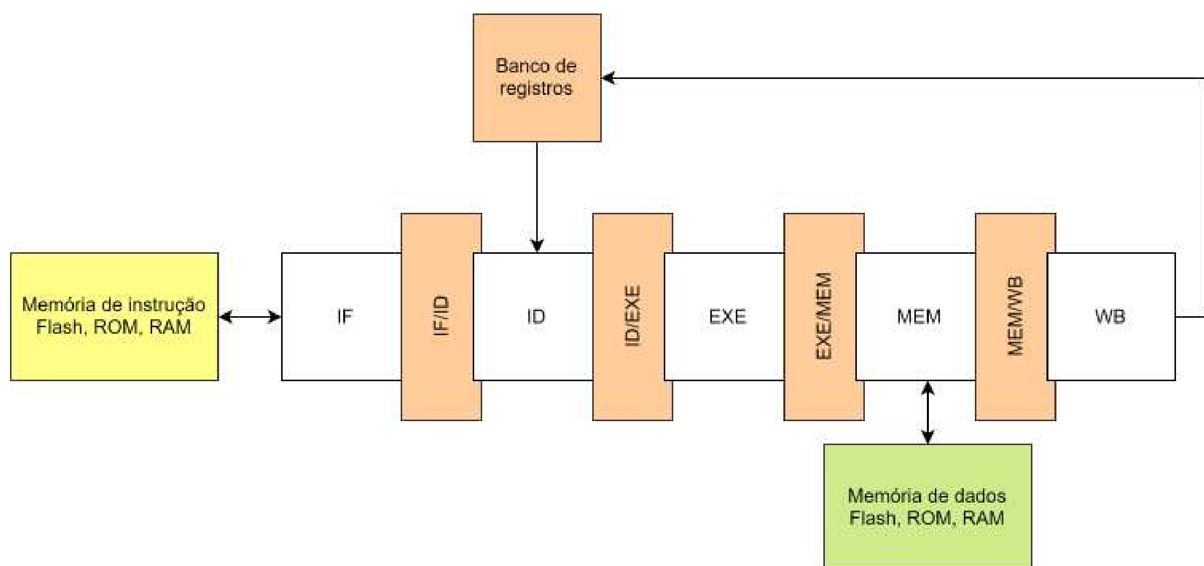


Figura 2 – Arquitetura do *pipeline* do processador de 5 estágios.

No entanto, se uma falha for injetada no circuito, os bits corrompidos são passados para a microarquitetura, que, por sua vez, os utiliza para executar operações no *pipeline* (como ilustra a Figura 3). Como resultado, as injeções de falhas em portas lógicas, memórias, *flip-flops*, etc., influenciarão a maneira como o sistema opera. As falhas podem causar:

- Corrupção de instruções;
- Corrupção de registros;
- Corrupção de dados;
- Corrupções na execução de cálculos;
- Corrupção do fluxo de controle.

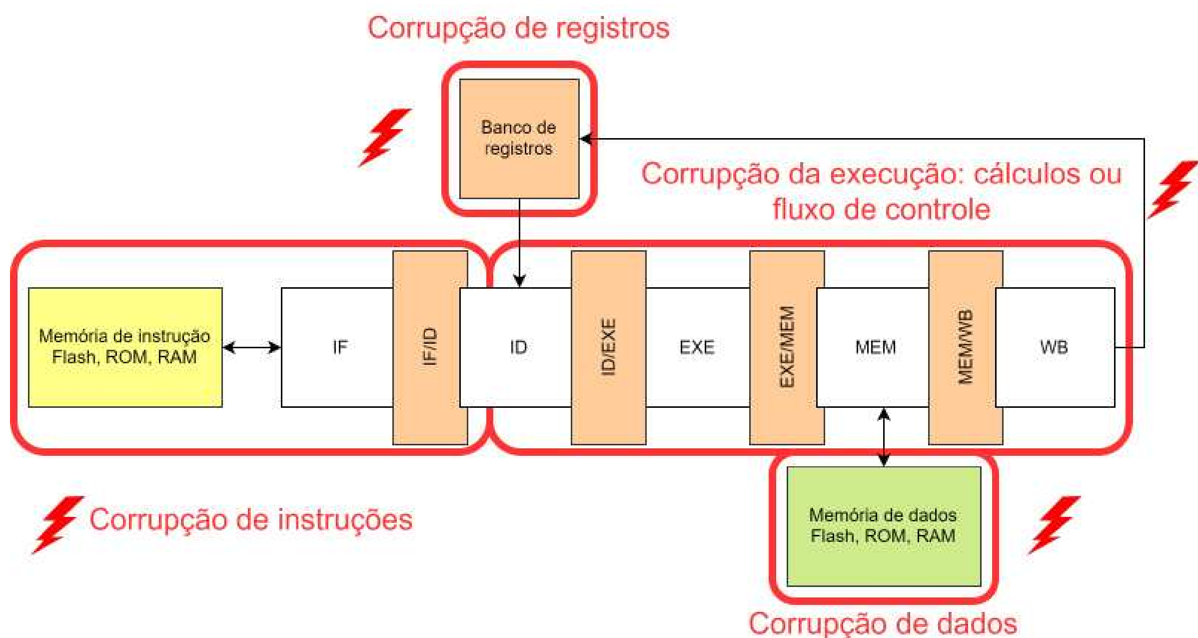


Figura 3 – Arquitetura de *pipeline* de processador de 5 estágios corrompida.

Dessa forma, a Figura 4 mostra como as operações corrompidas no nível do *hardware* são propagadas para o nível do *software* por meio da arquitetura de conjunto de instruções - Instruction Set Architecture (ISA). A ISA desempenha um papel fundamental como interface entre o *hardware* e o *software* do processador, pois define as instruções que o processador pode executar, sua codificação, os modos de endereçamento e os tipos de dados manipulados pelo processador. As corrupções podem afetar não apenas os dados, mas também o fluxo de controle do programa, permitindo que um atacante manipule o comportamento geral do sistema.

Dessa forma, as instruções corrompidas permitem que um atacante explore as vulnerabilidades de um sistema. Os objetivos do atacante são realizar uma ação para a qual ele não tem permissão. Dependendo do alvo, os objetivos podem ser:

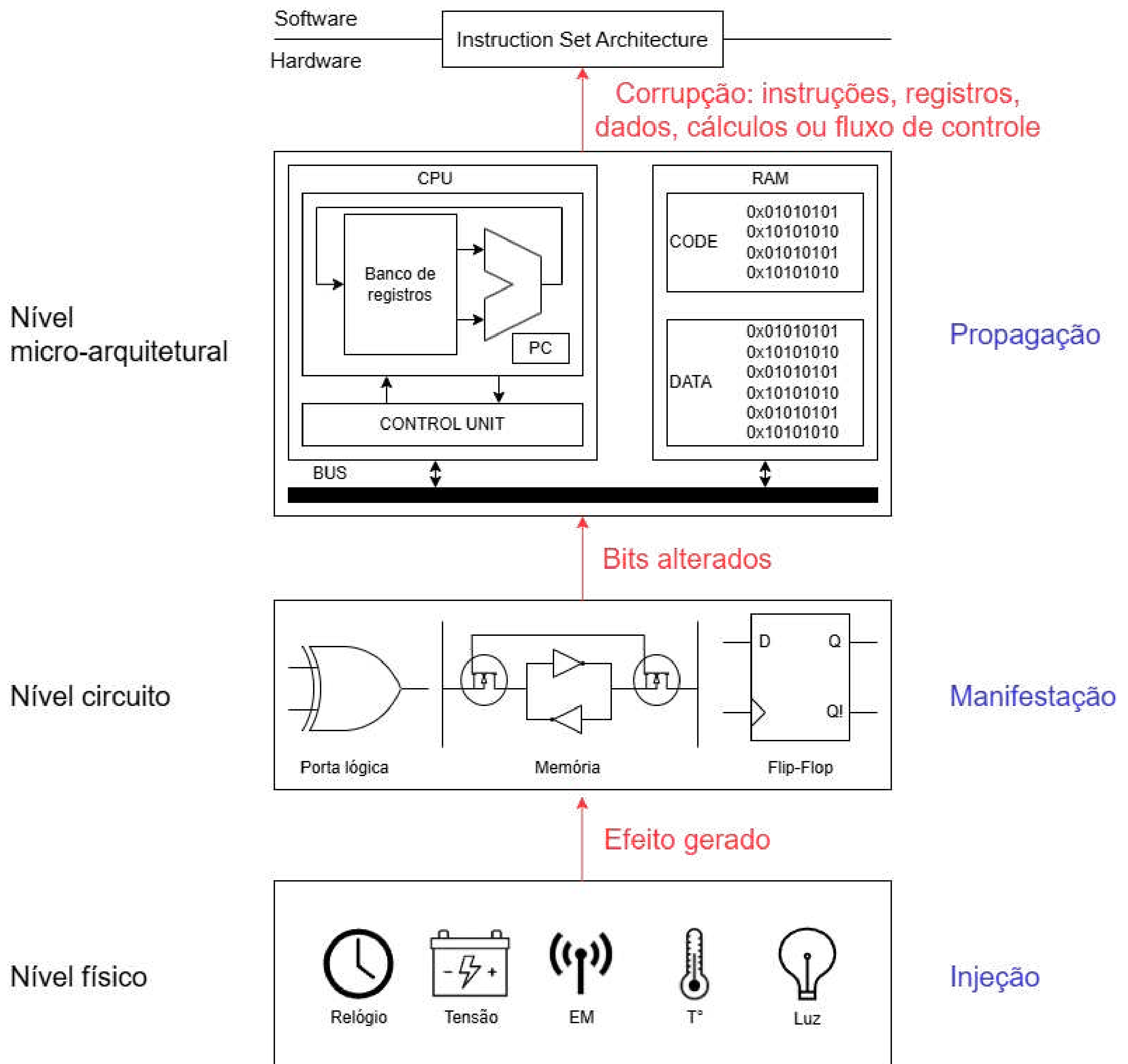


Figura 4 – Exemplificação de como a corrupção de dados a nível de *hardware* é transmitida para o nível do *software*.

- **Corrupção de dados confidenciais**: modificar ou corromper dados armazenados ou dados em trânsito;
- **Mau funcionamento do sistema**: interromper a operação normal do sistema para causar uma pane;
- **Comprometimento da segurança**: contornar mecanismos de segurança para obter acesso não autorizado.

Por exemplo, a Figura 5 mostra um chip encarregado por efetuar a verificação entre a senha inserida pelo usuário e a senha armazenada no sistema. A Figura 6 exemplifica um código de verificação entre duas senhas, logo se as senhas forem idênticas, o usuário terá acesso ao status de administrador, caso contrário o seu acesso será recusado. Na situação onde as senhas

não correspondem, espera-se receber um resultado negativo, i.e. o usuário não terá o status de administrador do sistema. Entretanto, se um ataque acontece durante a execução da verificação das senhas de modo que o resultado da comparação seja modificado, o usuário poderia obter um acesso não autorizado, o que não deveria acontecer.

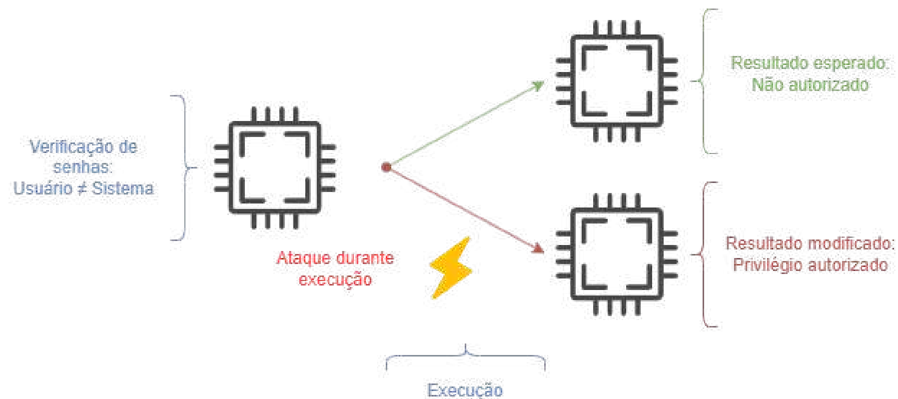


Figura 5 – Exemplo de como o ataque de injeção de falha de *hardware* pode contornar a verificação de senha.

```
1  #define FALSE 0
2  #define TRUE 1
3
4  int verifyPIN(char *cardPin, char *userPin, unsigned size){
5      unsigned i;
6
7      /********** Comparaison **********/
8      for (i = 0; i < size ; i++)
9          if (userPin[i] != cardPin[i])
10             return FALSE;
11
12     return TRUE;
13 }
```

Figura 6 – Código de verificação do PIN (Heydemann 2024).

Do ponto de vista da segurança, esses ataques representam uma grande ameaça aos sistemas embarcados, pois eles desempenham um papel fundamental na proteção de vários tipos de dispositivos e infraestruturas, como *smartphones*, dispositivos médicos, dispositivos de Internet of Things (IoT), cartões bancários etc. A falha desses sistemas pode, portanto, causar danos materiais, interrupções de serviços e riscos à segurança pública. Como resultado, há um desejo crescente de estudar as injeções de falhas e entender melhor seus efeitos para analisar a segurança do sistema ou desenvolver contramedidas.

1.2 Objetivos do trabalho

A missão desse trabalho é de realizar uma análise de segurança do processador de código aberto CV32E40S no modelo de atacante incluindo injeções de falhas. Para isso, a análise de segurança será realizada combinando várias abordagens, tais que a simulação de modelos Register Transfer Level (RTL) com auxílio de ferramentas como o Questasim ou Verilator, assim como métodos formais como μArchiFI. O objetivo é de propôr uma formalização de cada contramedida implementada no processador CV32E40S, e de qualificar e quantificar o ganho de segurança conseguido para cada contramedida, separadamente ou em combinação. Eis as especificações desse trabalho:

- Desenvolvimento de competências científicas e técnicas:
  - Realizar estudos bibliográficos de trabalhos relacionados a ataques por injeção de falhas em sistemas embarcados a nível de simulação: FDTC 2022 (Tollec *et al.* 2022), FMCAD 2023 (Tollec *et al.* 2023) e CHES 2024 (Tollec *et al.* 2024), e o contexto científico do trabalho;
  - Dominar os ambientes de simulação RTL;
  - Gerenciar a compilação e a execução de programas *bare metal* para arquiteturas Reduced Instruction Set Computer - Five (RISC-V) de 32 bits;
  - Estudar e simular a arquitetura do processador RISC-V CV32E40S e suas contramedidas.
- Trabalho de pesquisa:
  - Análise do processador CV32E40S e a contribuição de cada contramedida num modelo de atacante baseado em falhas (com base numa análise do código-fonte do processador e documentação);
  - Propor uma metodologia de análise da robustez para caraterizar a segurança fornecida por cada contramedida do CV32E40S.

2 ESTADO DA ARTE

2.1 Ferramentas de *software* utilizados

Este trabalho consiste em uma análise de segurança da arquitetura RISC-V, a qual é uma arquitetura de instruções que se diferencia das demais por ser aberta e gratuita. Em decorrência disto, o microprocessador CV32E40S está disponível em código aberto, o que permite realizar testes de segurança a nível de simulação. Logo, foi necessário utilizar diversos programas e ferramentas para atingir os objetivos propostos. Os principais programas empregados foram:

- **Podman**: uma alternativa de código aberto ao *Docker*, o *Podman* é utilizado para gerir contêineres, facilitando a implantação e a execução de ambientes de desenvolvimento específicos sem interferir no sistema hospedeiro, garantindo assim a consistência e a portabilidade da configuração (Podman 2024);
- **Git**: responsável para o controle de versões, permitindo acompanhar as alterações ao código e colaborar eficazmente com outros programadores, bem como gerir diferentes ramos de desenvolvimento;
- **GTKWave**: é uma ferramenta de visualização de formas de onda de código aberto utilizada para analisar sinais digitais gerados durante simulações de microprocessadores. Pode ser utilizada para visualizar transições de sinais e cronogramas e para verificar o comportamento funcional de circuitos digitais (GTKWave 2024);
- **Questasim**: um simulador de Hardware Description Language (HDL), uma ferramenta da *Mentor Graphic*, utilizado para simular e verificar o design de microprocessadores. Suporta as linguagens Verilog, VHDL e SystemVerilog, e oferece funcionalidades avançadas de depuração e verificação para garantir que o projeto cumpre as especificações;
- **Verilator**: é um simulador HDL de código aberto utilizado para converter código Verilog em código C++ e efetuar simulações rápidas (Verilator 2024);
- **Yosys**: é um *software* de síntese utilizado para transformar o código HDL numa lista de rede optimizada para a concepção de FPGA. Pode ser utilizado para otimizar a lógica, verificar a conetividade do circuito e preparar o projeto para implementação em dispositivos FPGA. O *Yosys* suporta a linguagem Verilog na sua versão de código aberto e oferece suporte parcial para SystemVerilog numa versão paga (YosysHQ 2024);
- **sv2v**: é um conversor de código aberto de SystemVerilog para Verilog, facilitando o uso

de certos recursos avançados do SystemVerilog em ferramentas que apenas suportam Verilog. Este aspecto facilita a integração e a compatibilidade entre diferentes ferramentas e ambientes de desenvolvimento (Snow 2024);

- **Compilador RISC-V fornecido pela Embescom**: um compilador especialmente concebido para o microprocessador RISC-V, capaz de traduzir código escrito em C e assembler em instruções que podem ser executadas pelo microprocessador (Embecosm 2024).

2.2 Workflow uArchiFI

De acordo com os trabalhos da equipe de segurança CEA-LETI, o qual podem ser encontrados em FDTC (Tollec *et al.* 2022) e FMCAD (Tollec *et al.* 2023), os atacantes FIA podem explorar, entre outras coisas, falhas no mecanismo de *forwarding*, que permite que valores calculados anteriormente sejam recuperados e reinjetados no *pipeline*. Além do mecanismo de *forwarding*, outras vulnerabilidades foram identificadas, como falhas no *prefetch buffer*, que podem ter vários efeitos prejudiciais no *software*, como a repetição imediata de instruções presentes no *prefetch buffer*, a execução desordenada de instruções ou a corrupção do alvo da próxima ramificação.

Houve, portanto, a necessidade de modelar e analisar as falhas, considerando tanto o *software* quanto o *hardware*, a fim de compreender melhor seus efeitos. Esses métodos podem contribuir para evidenciar a importância dos mecanismos de microarquitetura na análise do impacto das falhas sobre a segurança do sistema.

Para atender aos requisitos acima, segundo o artigo FMCAD (Tollec *et al.* 2023), foi proposto e desenvolvido o programa μArchiFI, mostrado na Figura 7, um *workflow* para a realização de uma análise formal de um sistema completo composto de *hardware*, *software* e componentes com injeção de falhas.

O μArchiFI baseia-se na ideia de verificação formal, que consiste em criar modelos matemáticos de sistemas e usar técnicas automatizadas para provar ou identificar contraexemplos às propriedades desses modelos. Em vez de testes e simulações, que incluem a possibilidade de não abranger todos os estados e restrições possíveis, a verificação formal usa técnicas de sondagem, como a Bounded Model Checking (BMC), para verificar se uma propriedade formal é satisfeita.

A BMC é uma técnica de verificação formal usada para detectar erros em sistemas sequenciais (como circuitos digitais) dentro de um limite finito de transições de estado. Em vez

de explorar todos os estados possíveis do sistema, o que pode ser impraticável para sistemas complexos devido ao crescimento exponencial do espaço de estados, o BMC examina apenas uma sequência finita de transições a partir do estado inicial.

Um sistema de transição material se entende como um modelo matemático que representa o comportamento sequencial dos circuitos digitais. Ele é definido por um conjunto de possíveis estados do circuito (S), um conjunto de estados iniciais (So), um conjunto de entradas (X) e uma função de transição que associa cada par de estados e de entradas a um novo estado ($T : S \rightarrow S$). Esse modelo descreve como o estado do circuito evolui no tempo em resposta a suas entradas, o que permite analisar e verificar o comportamento do circuito.

Embora o BMC tenha um limite finito de transições de estado, ele ainda exige muitos recursos computacionais para realizar a verificação formal. Para otimizar esse processo, são oferecidas duas opções:

- **Sandboxing**: é uma técnica usada para restringir o comportamento do código do *software* em um ambiente controlado e limitado. Para isso, são impostas restrições ao PC, de modo que ele só possa assumir valores específicos derivados de uma análise estática do código binário. Essas restrições impedem a exploração de caminhos de execução indesejados, melhorando assim a segurança e a estabilidade do sistema. Entretanto, ao impor essas restrições, pode-se perder a capacidade de verificar exaustivamente todas as condições operacionais possíveis do programa, o que torna a técnica mais adequada para descobrir vulnerabilidades do que para garantir a robustez absoluta do sistema;
- **Concretização**: é uma técnica usada para gerenciar o sistema de transição, dividindo-o em subgrupos que codificam diferentes caminhos de execução do programa. Cada subgrupo pode representar um caminho específico que o programa pode seguir durante a execução. Isso reduz a complexidade da verificação, pois as partes do sistema de transição que são processadas são menores e mais fáceis de gerenciar.

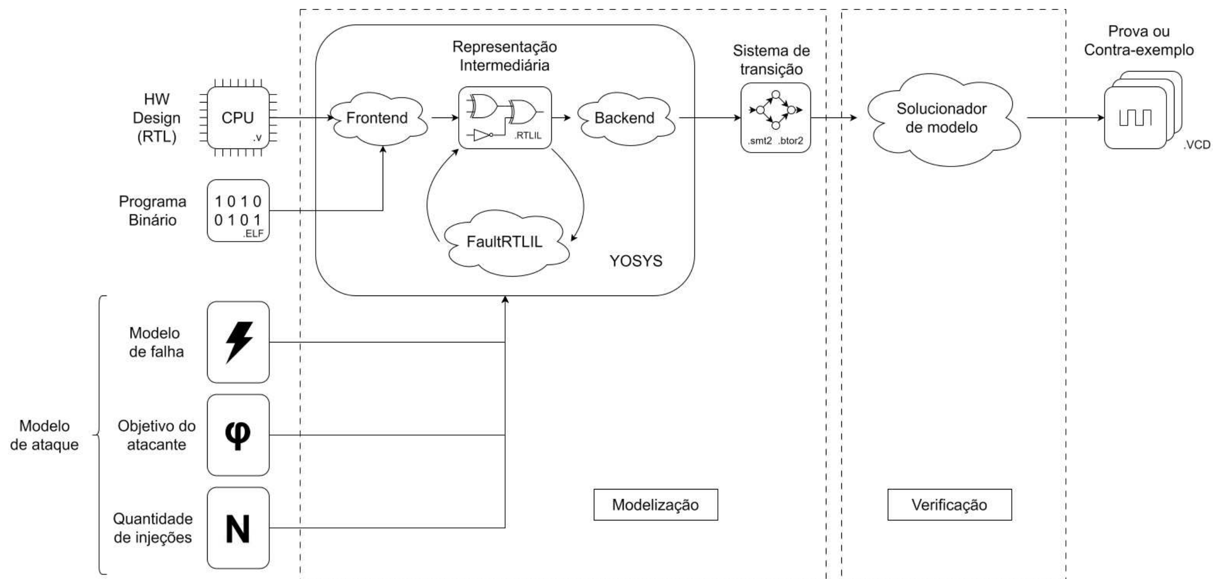


Figura 7 – Arquitetura e rede de ferramentas de verificação μArchiFI.

O μArchiFI recebe como entrada:

- A descrição do *hardware* do microprocessador escrita na linguagem Verilog;
- O programa de *software* binário no qual o usuário deseja que o microprocessador seja executado;
- O modelo de ataque, no qual o usuário define o objetivo, a quantidade máxima de falhas e o modelo de falha que corresponde ao local da injeção, o intervalo de tempo a ser aplicado e o efeito da falha.
  - Para essa proposta, o μArchiFI contém um modelo integrado ao sistema chamado FaultRTLIL. A partir dessa passagem, é possível definir o local de injeção, o intervalo de tempo a ser aplicado e o efeito da falha, que são: set, reset, flip, diff ou um valor especificado;
  - O usuário também pode definir o objetivo do ataque, especificando-o na descrição do *hardware* por meio de asserções do SystemVerilog.

A partir dessas entradas, o μArchiFI usa o *software* de código aberto Yosys para gerar o arquivo do sistema de transição, pois ele pode analisar e traduzir uma especificação de *hardware* em linguagens formais, permitindo que o foco seja a integração automatizada de um modelo de ataque no sistema.

A passagem FaultRTLIL atua na linguagem de representação intermediária RTL Intermediate Language (RTLIL) , integrando entradas definidas pelo usuário no sistema, substituindo o(s) fio(s) de destino da conexão por multiplexador(es) no circuito para emular falhas.

A Figura 8 mostra como isso funciona, onde o sinal sig corresponde ao sinal origi-

nal/alvo conectado ao multiplexador que o usuário deseja modificar entre as opções disponíveis (bit – set, bit – reset, flip, diff ou um valor especificado), que será, portanto, o sinal modificado sig_fault. O sinal de seleção sig_sel determina qual sinal será transmitido para a saída. Com base no intervalo de tempo determinado pelo usuário, o sinal de seleção selecionará o sinal modificado, enquanto fora desse intervalo ele selecionará o sinal original. O sinal de contagem sig_cnt é usado para controlar o número máximo de falhas a serem injetadas.

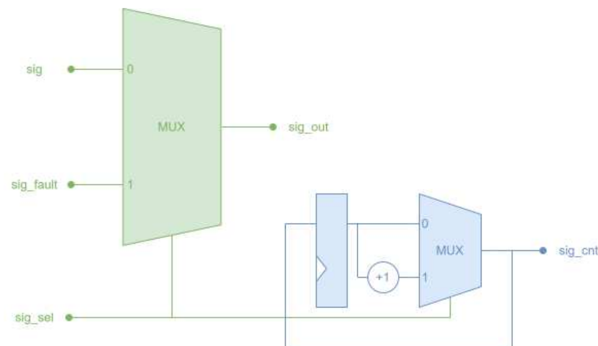
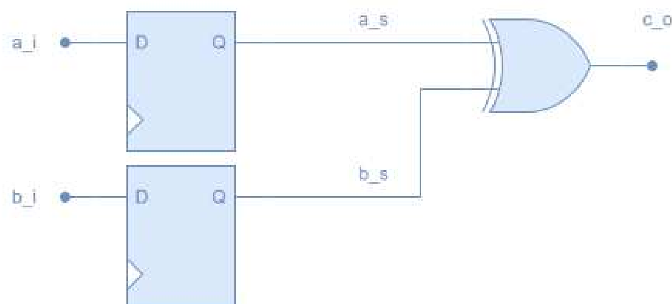
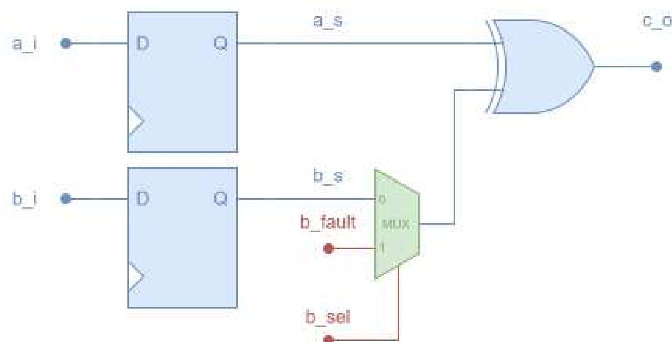


Figura 8 – Emulação de falhas por meio de multiplexadores.

Por exemplo, a Figura 9a mostra parte de um circuito que executa uma operação XOR, e a Figura 9b mostra o mesmo circuito com um multiplexador substituindo a conexão do fio b_s. Dessa forma, o multiplexador permite que a injeção de falha seja emulada usando o sinal b_fault a partir do sinal de seleção b_sel.



(a) Circuito original.



(b) Circuito modificado após a execução da passagem FaultRTLIL.

Figura 9 – Exemplo do uso da passagem FaultRTLIL.

O arquivo gerado conterá o lado do *hardware*, o lado do *software* e o modelo de ataque integrado ao sistema. Deve-se observar que o μArchiFI preserva todos os nomes de sinais e registros retirados da descrição de *hardware* Verilog, permitindo que o usuário selecione com precisão os locais a serem atacados.

A Yosys pode transformar uma descrição de *hardware* em um sistema de transição de *hardware*, com suporte aos formatos formais smt-lib e btor2.

- **smt-lib**: a *Satisfiability Modulo Theories Library* é um padrão para representar problemas de satisfatibilidade de módulo em lógica de primeira ordem com teorias específicas, como aritmética, matrizes, vetores de *bits* etc. Esse formato é amplamente usado em verificação formal, *model checking* e em solucionadores Satisfiability Modulo Theories (SMT) gerais, como Z3, CVC4, etc. Um arquivo smt-lib contém declarações de variáveis, definições de funções e restrições que devem ser satisfeitas.
- **btor2**: Bit-Vector Term Representation é um formato usado para descrever circuitos e sistemas de transição na forma de vetores de *bits*. Esse formato foi projetado para uso com ferramentas de verificação formal, como o Boolector, um solucionador de SMT . Um arquivo btor2 contém expressões textuais que descrevem variáveis, operadores e transições de sistema, facilitando a análise e a solução com ferramentas automáticas.

Uma vez que o usuário tenha definido as entradas, o μArchiFI gerará um sistema de transição no formato smt-lib ou btor2 do microprocessador, incluindo falhas definidas pelo script FaultRTLIL. Para verificar esses arquivos gerados, o μArchiFI fornece três comandos principais: yosysmc, ponoCentaur 2024 e btormcBoolector 2024, todos os quais usam a técnica BMC.

- **yosysmc**: é uma ferramenta de verificação formal integrada ao Yosys, uma estrutura de síntese para Verilog. O yosysmc é particularmente eficaz para verificar propriedades definidas em um arquivo Verilog, usando o formato smt-lib;
- **pono**: este é um verificador formal especializado na análise de modelos escritos em btor2. Ele usa a técnica BMC para explorar os diferentes estados possíveis de um sistema, procurando contraexemplos para as propriedades ou afirmações especificadas. É uma ferramenta flexível que suporta várias estratégias de verificação, o que a torna ideal para sistemas complexos que exigem análise detalhada;
- **btormc**: este é um comando dedicado à verificação de sistemas de transição descritos no btor2. Esse verificador BMC concentra-se em vetores de bits, o que o torna muito eficiente para modelos de circuitos lógicos. O btormc verifica as propriedades do modelo

procurando contraexemplos que indicariam uma violação das propriedades especificadas.

A ferramenta de verificação de modelo escolhida executará a verificação, procurando um contraexemplo em relação aos objetivos definidos pelo modelo de ataque ou pelas asserções do SystemVerilog adicionadas ao arquivo Verilog. Assim, se o solucionador conseguir encontrar um contraexemplo, ele gerará um arquivo Value Change Dump (VCD) que indica precisamente onde a falha foi injetada e quando o objetivo do atacante foi alcançado. Por outro lado, se o solucionador não encontrar nenhum contraexemplo, isso significa que ele não detectou nenhuma violação das propriedades definidas, o que sugere que o *hardware* e o *software* conseguiram lidar com a falha sem que o atacante conseguisse atingir seus objetivos.

3 DESENVOLVIMENTO

Inicialmente, após a análise do trabalho, descrito na Seção 2, foram utilizados estudos de caso presentes nas publicações para familiarização com o ambiente de desenvolvimento. Em seguida, foi realizada a simulação do microprocessador CV32E40P e a geração de pequenos programas por meio da cadeia de compilação *bare metal* para microprocessadores RISC-V de 32 bits. Por fim, a robustez do microprocessador foi avaliada por meio da injeção de falhas com a ferramenta μArchiFI.

3.1 Apresentação do CV32E40P

No artigo FMCAD (Tollec *et al.* 2023), usaram a ferramenta μArchiFI para avaliar a robustez do microprocessador CV32E40P. O CV32E40P é um núcleo RISC-V de 32 bits, pequeno, eficiente e de execução sequencial, desenvolvido pelo OpenHWGroup (OpenHW Group 2024). Ele é equipado com um *pipeline* de 4 estágios (IF, ID, EXE e WB) que implementa a arquitetura do conjunto de instruções RV32IM[F|Zfinx]C, bem como extensões PULP personalizadas destinadas a melhorar a densidade do código, o desempenho e a eficiência energética (OpenHW Group 2024), como mostrado pela Figura 10. Os conjuntos de instruções implementados são:

- **I**: conjunto de instruções baseado em números inteiros;
- **M**: extensão padrão para multiplicação e divisão de números inteiros;
- **C**: extensão padrão para instruções compactadas;
- **F**: cálculo de ponto flutuante de precisão única usando registradores F;
- **Zfinx**: cálculo de ponto flutuante de precisão única usando os registradores X.

O microprocessador CV32E40P não possui contramedidas de segurança integradas, seja em seu conjunto de instruções ou em sua arquitetura de *hardware*.

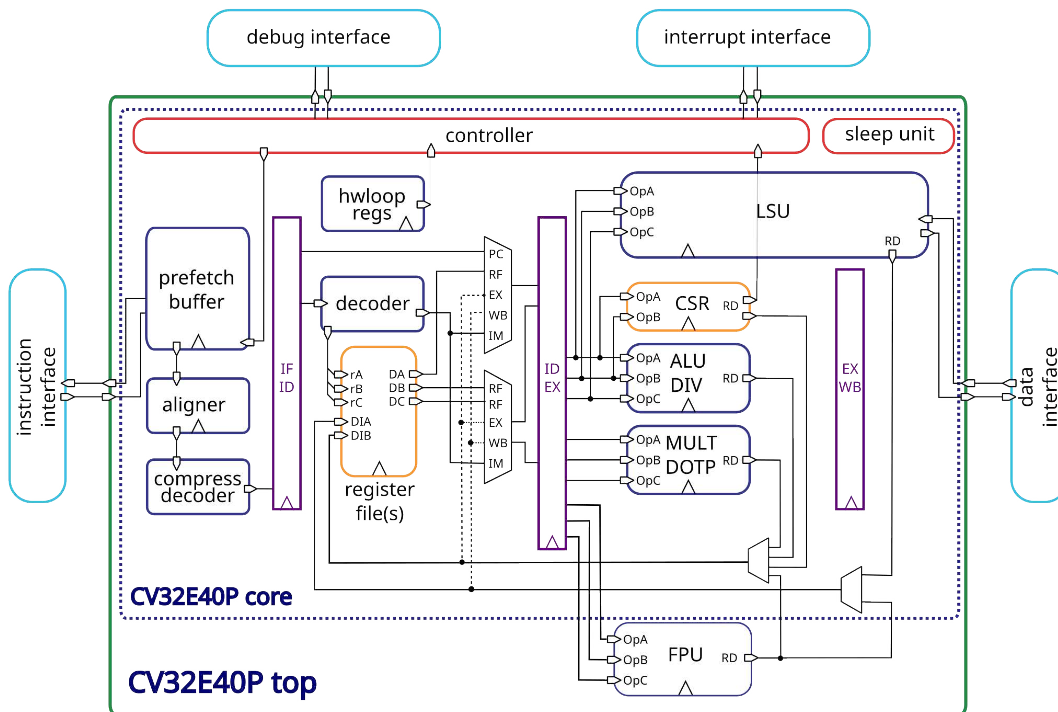


Figura 10 – Esquemático do núcleo CV32E40P.

3.1.1 Geração de códigos binários

Para usar o microprocessador CV32E40P, é necessário um compilador compatível com a arquitetura RISC-V, que suporta o conjunto de instruções ISA. Ele executa as instruções de código de máquina depois que elas são carregadas na memória. Consequentemente, é importante ter um ambiente de programação capaz de converter um arquivo escrito em C ou em linguagem *assembly* em um arquivo binário, que é então executado pelo microprocessador.

O CORE-V-VERIF, fornecido pelo OpenHWGroup (OpenHW Group 2024), é um projeto em desenvolvimento para verificar os seguintes microprocessadores: CV32E40P, CV32E40S, CV32E40X e CVA6. Esse projeto fornece um ambiente de programação que alinha o código do *software* com o código RTL do microprocessador, conforme mostrado na figura 11, onde:

- **testcase.c**: representa o código do *software* escrito em C ou em linguagem assembly;
- **crt0.s**: esse script executa o mínimo de *bare metal* necessário para executar um programa em C;
- **link.ld**: o código-fonte é compilado em código-objeto e, em seguida, o script link.ld

combina esses arquivos-objeto em um arquivo executável binário;

- **GCC Compile**: corresponde ao compilador usado para compilar o código-fonte em um arquivo binário;
- **elf2hex**: corresponde à conversão do arquivo binário em um arquivo hexadecimal;
- **mm_ram**: corresponde à memória de instruções;
- **dp_ram**: corresponde ao local de todas as instruções;
- **VP**: corresponde aos periféricos virtuais usados para simulação;
- **addr, data, irq**: correspondem aos sinais que se comunicam entre a memória e o microprocessador (endereços, dados, sinais de interrupção);
- **CV32E40P CORE RTL** : corresponde ao *hardware* do microprocessador.

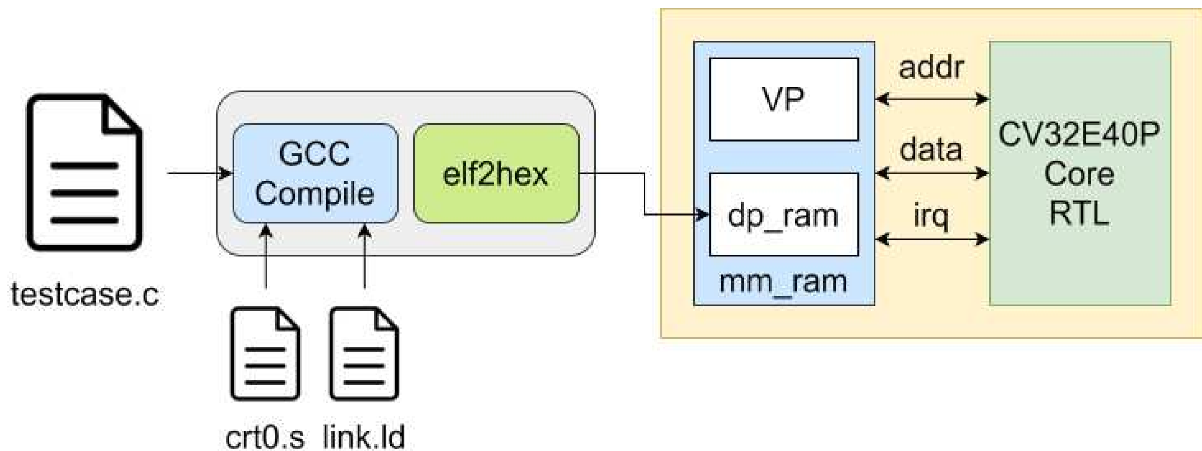


Figura 11 – Ambiente de programação para o CV32E40P.

O ambiente de programação utilizado requer uma máquina com sistema operacional Linux, Python 3 com as dependências especificadas no manual, e o compilador RISC-V fornecido pela Embecosm, que inclui os conjuntos de instruções utilizados pelo microprocessador em questão. Adicionalmente, é necessário configurar variáveis globais para definir as arquiteturas a serem utilizadas, o simulador com suporte à Universal Verification Methodology (UVM) versão 1.2, o kernel a ser simulado, entre outros parâmetros.

Como mencionado acima, o ambiente de trabalho do CORE-V-VERIF usa a biblioteca UVM , que é uma metodologia usada no setor para verificar projetos de *hardware*, como microprocessadores, usando a linguagem SystemVerilog. Ela oferece uma estrutura modular e reutilizável que facilita a criação de bancos de testes complexos, promovendo a abstração e a automação do processo de verificação. O UVM permite a geração automática de estímulos de teste e a verificação dos resultados esperados, melhorando a cobertura e a qualidade do projeto.

Uma vez instalado o CORE-V-VERIF e seguidas todas as instruções, foi possível simular o núcleo P com o programa binário. Portanto, foi decidido usar um código de verificação cuja finalidade é comparar dois códigos PIN armazenados na memória e, em seguida, validar ou não a autorização de acesso do usuário. Como se trata de um protocolo de segurança, é imperativo que a verificação não seja corrompida, para que o processo seja executado corretamente.

A coleção de códigos públicos Fault Injection and Simulation Secure Collection (FISSC) (Dureuil *et al.* 2016) oferece uma coleção de códigos C com contramedidas contra injeções de falhas associadas a cenários de ataque. Várias versões do código verifyPIN foram estudadas durante o curso. O texto a seguir baseia-se na sétima versão do FISSC (ver Figura 12) para representar o lado do *software*, que contém os seguintes recursos:

- **Reforço da variável booleana**: utiliza variáveis booleanas com valores fixos 0xAA e 0x55 para evitar a utilização de valores muito sensíveis a FIA;
- **Loop de tempo fixo**: garante que o tempo de execução da verificação do PIN não revela o número de caracteres corretos, evitando assim ataques temporais. No contexto das FIA, isto evita sair momentaneamente de um ciclo;
- **Diminuir primeiro**: isto diminui o contador de tentativas antes de prosseguir com a verificação do PIN. Isto garante que cada tentativa é corretamente contada. No contexto das FIA, isto pode reduzir a necessidade de manipular o contador de novas tentativas para ganhar hipóteses adicionais;
- **Dupla chamada**: esta proteção consiste em verificar duas vezes o código PIN para detetar eventuais erros ou manipulações. No caso de um ataque de FIA, a dupla verificação identifica inconsistências e garante que o processo de verificação é fiável, apesar das tentativas de manipulação.

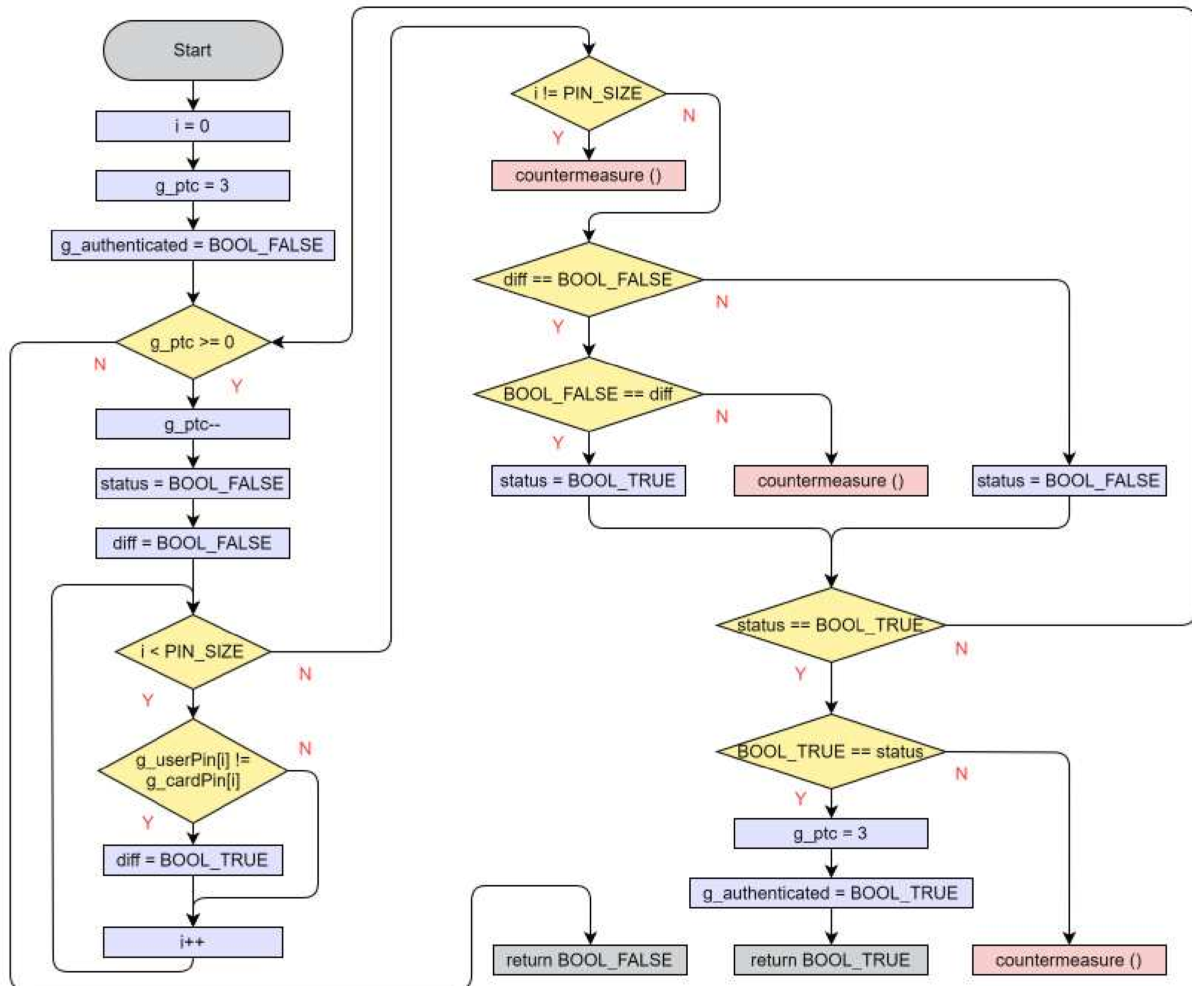


Figura 12 – Fluxograma da verificação de PINs - função *verifyPIN* versão 7.

Quando um erro é detectado, a função *countermeasure* é chamada para acionar o sinal *g_countermeasure*, indicando ao sistema a presença de uma possível anomalia. Além disso, as variáveis g_userPin e g_cardPin são inicializadas previamente na função *initialize.c* (consulte o apêndice A).

Foi usado o parâmetro *-Og* para compilar o código sem otimização, de modo que o compilador não remova nenhuma contramedida implementada, pois a função *verifypin_v7* usa redundâncias em seu código para garantir a segurança do sistema.

3.1.2 Simulação do microprocessador + programa binário

Após a compilação do código-fonte, o arquivo binário gerado é carregado na memória de instruções para que o microprocessador possa executar as instruções desejadas. Como mostra a Figura 2, o microprocessador inicializa o carregamento de instruções no estágio IF e, em seguida, as informações são direcionadas para os estágios subsequentes.

Para determinar quando um comando é executado, pode-se usar a janela de traços do simulador ou consultar o arquivo *log* gerado pelo simulador. Com o Questasim, pode-se ver as formas de onda do microprocessador executando as instruções do código binário *verifypin_v7.c*. Também é possível ver o tempo, o ciclo de *clock*, as instruções no código de máquina, o PC e os registros usados no arquivo *log* gerado pelo ambiente CORE-V-VERIF. Isto permite verificar as instruções executadas pelo microprocessador, comparando-as com o arquivo de código *verifypin_v7.c* desassemblado.

3.1.3 Conversão do hardware

Como o conjunto de ferramentas μArchiFI usa o *software* Yosys que, por sua vez, só lê arquivos escritos na linguagem Verilog em sua versão *open-source*, é necessário converter os arquivos RTL e os arquivos de memória escritos na linguagem SystemVerilog em Verilog. Para fazer isso, utilizou-se a ferramenta *sv2v*.

3.1.4 Avaliaçao da robustez do CV32E40P

A simulação do microprocessador com o programa binário tem como objetivo principal verificar se os comandos são executados corretamente e se não há erros durante sua execução. Após essa etapa de verificação, realiza-se uma avaliação funcional com o intuito de identificar possíveis falhas ou vulnerabilidades que possam surgir durante a execução do programa.

Para o código *verifypin_v7.c*, é possível tentar injetar uma falha durante o intervalo de atribuição de valores às variáveis usadas pelo código para que o atacante possa ter o estado autenticado. Isso ocorre durante a execução da função *verifyPIN* (não confunda o nome da função com o nome do código). O objetivo do atacante será, portanto, contornar o mecanismo de autenticação sem acionar as contramedidas de *software* no final do programa:

φ_P := (authenticated $\wedge \neg$ software_alert)

Para isso, precisa-se inicialmente encontrar a posição na memória onde estão armazenados os dados das variáveis g_authenticated, g_countermeasure, g_userPin e g_cardPin, que são inicializadas na função *initialize* no código *verifypin_v7.c*. Usando a função de desmontagem, é possível examinar o conteúdo do arquivo binário gerado após a compilação do código *verifypin_v7.c*, o que permite localizar os endereços onde os valores de registro estão armazenados, incluindo g_authenticated, g_countermeasure, g_userPin e g_cardPin. Observando o arquivo de

código desmontado no apêndice A, os endereços de cada variável são:

Variável	Endereço hexadecimal	Endereço binário	Valor armazenado
g_userPin[0]	0x134	1_0011_0100	5
g_userPin[1]	0x135	1_0011_0101	6
g_userPin[2]	0x136	1_0011_0110	7
g_cardPin[0]	0x138	1_0011_1000	1
g_cardPin[1]	0x139	1_0011_1001	2
g_cardPin[2]	0x13A	1_0011_1010	3
g_countermeasure	0x13B	1_0011_1011	0
g_authenticated	0x13D	1_0011_1101	0x55 (Falso)

Tabela 1. Endereços e valores de cada variável usada no código *verifypin_v7.c*.

Como os números (ou PINs) são diferentes, o usuário não deve ter acesso ao estado autenticado. Isso significa que a variável g_authenticated não pode assumir o valor 0xAA, que representa o estado verdadeiro. Se a variável g_authenticated tiver o valor 0xAA, a variável g_countermeasure deverá ser definida como 1 (valor verdadeiro) para indicar que o sistema foi atacado durante a operação, porque as variáveis g_userPin e g_cardPin contêm valores diferentes. A Tabela 2 mostra as diferentes opções de variáveis e suas interpretações. Os testes realizados concentraram-se no cenário representado pela terceira linha da Tabela, o qual corresponde ao objetivo do atacante φ_P.

g_authenticated	*g_countermeasure*	Interpretação
0x55 (Falso)	0 (Falso)	sem acesso e ataque não detectado
0x55 (Falso)	1 (Verdadeiro)	sem acesso e ataque detectado
0xAA (Verdadeiro)	0 (Falso)	acesso bem-sucedido e ataque não detectado
0xAA (Verdadeiro)	1 (Verdadeiro)	acesso bem-sucedido e ataque detectado

Tabela 2. Possibilidades e suas interpretações para as variáveis g_authenticated e g_countermeasure.

Após a descoberta dos endereços e a definição da lógica a ser testada, foram especificadas as propriedades por meio de asserções em SystemVerilog, alinhadas ao objetivo do atacante φ_P, conforme apresentado a seguir. Adicionalmente, foram incluídas asserções relativas aos códigos PIN, a fim de garantir que permanecessem distintos ao longo de toda a verificação.

```
assign PIN1 = mem['b100110100] != mem['b100111000];
assign PIN2 = mem['b100110101] != mem['b100111001];
assign PIN3 = mem['b100110110] != mem['b100111010];
assume property(PIN1);
assume property(PIN2);
assume property(PIN3);
assign unauthenticated = (mem['b100111101] != 8'haa) ||
                         (mem['b100111011] != 8'h0);
assert property(unauthenticated);
```

Figura 13 – Propriedades estabelecidas para o CV32E40P verificar.

Observando a simulação no Questasim, a função *initialize* termina no ciclo 69 e a função *verifyPIN* começa no ciclo 72. As verificações das variáveis *diff* e *status* ocorrem após 60 ciclos de *clock* a partir do início da função *verifyPIN* e terminam no ciclo 80. O objetivo é interferir nas alocações responsáveis por determinar a autenticação do usuário, a fim de avaliar a capacidade do conjunto *hardware* + *software* em lidar com tentativas de adulteração.

Para isso, foi definido que o modelo de ataque injetará uma única falha na lógica sequencial do caminho de controle do microprocessador durante o intervalo de *clock* 60 a 80, porque este intervalo inclui a instrução de comparação que determina se o usuário será ou não autenticado (esse é o último bloco de comparação no fluxograma da figura 12). Foi selecionado todos os fios de saída dos registros entre 1 e 8 bits de largura para aplicar uma injeção de efeito diferente do valor correto durante esse intervalo. Dessa forma, o modelo de ataque visou 102 dos 179 registros do microprocessador.

```
select t:*ff* %co t:*ff* %d s:1:7 %i
fault_rtlil -effect diff -timing 60:80 -cnt 1
```

Figura 14 – Comandos Yosys utilizados para injetar falhas no CV32E40P.

A Figura 15 ilustra como é verificado a robustez do *hardware* e do *software*:

- Usando o Yosys, foi simulado o CV32E40P executando instruções do código *verifypin_v7.c* para as funções *main* e *initialize*, ou seja, foram simulados 72 ciclos de *clock*;
- Para garantir que os PINs sejam diferentes, foi substituído os valores de g_userPin e g_cardPin por valores simbólicos após sua inicialização;
- Em seguida, a passagem FaultRTLIL foi usada para adicionar multiplexadores ao circuito a fim de emular a injeção de falhas (consulte a Figura 14);
- Uma vez que o microprocessador contém os dados registrados pelas instruções executadas

e contém os multiplexadores adicionados ao circuito, o Yosys gera arquivos de sistema de transição no formato .smt2 e .btor2;

- No final, utiliza-se solucionadores como yosysmc, pono e btormc para verificar os arquivos do sistema de transição gerados e determinar se as propriedades definidas (consulte a figura 13) foram respeitadas.

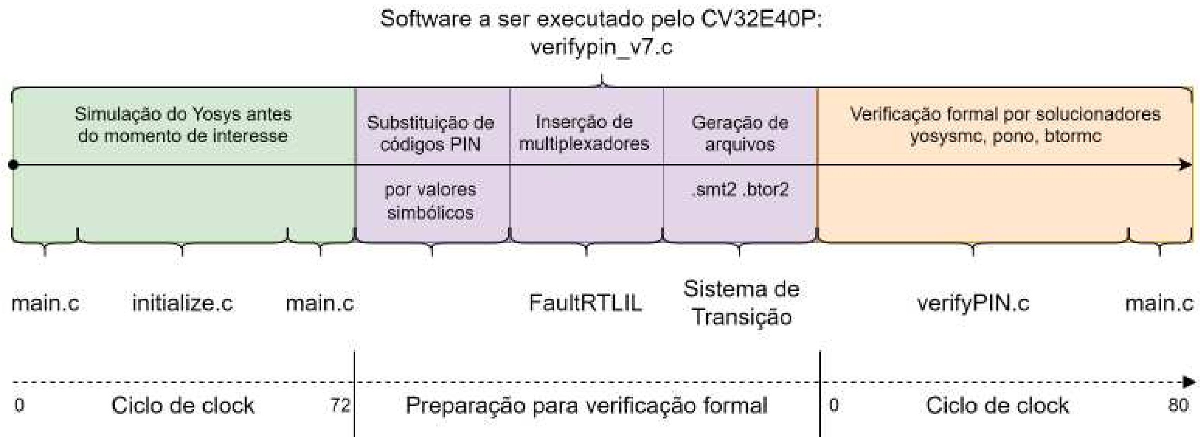


Figura 15 – Cronologia para a avaliação da robustez do CV32E40P + *verifypin_v7.c*.

Depois de compilar o *software*, converter o *hardware* de SystemVerilog para Verilog e definir as propriedades a serem verificadas, foi executado a verificação formal. A Figura 16 mostra o resultado obtido usando o solver pono. A interpretação do resultado obtido é que o solucionador detectou que uma das propriedades definidas não foi respeitada, o que indica que o atacante conseguiu obter o estado autenticado sem acionar o alarme de *software* esperado, apenas com uma injeção de falha. Portanto, o atacante pode tirar proveito dessa vulnerabilidade para obter acesso não autorizado aos procedimentos de verificação do PIN. Todos os solucionadores detectaram a vulnerabilidade no ciclo 70 a partir do início da função *verifyPIN*.

Usando o *software* GTKWave, é possível observar os contraexemplos fornecidos pelos solucionadores que permitem ao usuário encontrar o local exato da falha que leva à vulnerabilidade. O sinal explorado foi o sinal alu_vec_mode_ex_o, que está bem listado na Tabela II do artigo FDTC (Tollec *et al.* 2022).

A publicação FMCAD (Tollec *et al.* 2023) apresenta três estudos de caso, dentre os quais é abordado, neste trabalho, o caso referente à robustez de *software*. Para essa finalidade, foi configurado um ambiente de desenvolvimento voltado à geração de programas binários e sua execução no microprocessador CV32E40P, permitindo a reprodução dos resultados apresentados na Seção VI – *Evaluation* da referida publicação.

```
BMC checking at bound: 68
  BMC reached_k = 67, i = 68
  BMC adding transition for j-1 = 67
  BMC adding bad state constraint for j = 68
  BMC check at bound 68 unsatisfiable

BMC checking at bound: 69
  BMC reached_k = 68, i = 69
  BMC adding transition for j-1 = 68
  BMC adding bad state constraint for j = 69
  BMC check at bound 69 unsatisfiable

BMC checking at bound: 70
  BMC reached_k = 69, i = 70
  BMC adding transition for j-1 = 69
  BMC adding bad state constraint for j = 70
  BMC check at bound 70 satisfiable
  BMC saving reached_k_ = 69
  BMC get cex upper bound: lower bound = 70, upper bound = 70
    BMC get cex upper bound, checking value of bad state
       constraint j = 70
    BMC get cex upper bound, found at j = 70
  BMC permanently blocking interval [start,end] = [71,70]

  BMC binary search, cex found in interval [reached_k+1,
     upper_bound] = [70,70]
  BMC interval has length 1, skipping search for shortest cex
```

Figura 16 – Resultado da verificação formal gerada pelo solucionador pono.

3.2 Desenvolvimento do ambiente de trabalho - CV32E40S

O microprocessador CV32E40S, fornecido de maneira gratuita no seu site oficial (OpenHW Group 2024), começou como um derivado do microprocessador CV32E40P e continua em desenvolvimento. É um pequeno núcleo RISC-V de 32 bits equipado com um *pipeline* de 4 estágios (IF, ID, EXE e WB), como exibe a Figura 17. Ele implementa os conjuntos de instruções CV32E40S e os conjuntos abaixo:

- **Zicsr**: Instruções de registro de controle e status;
- **Zifencei**: Instruções Fetch Fence;
- **Zca, Zcb, Zcmp, Zcmt**: Subconjunto da extensão de redução de tamanho de código Zc padrão;
- **Zba, Zbb, Zbc, Zbs**: Manipulação de bits;
- **Xsecure**: Extensões de segurança.

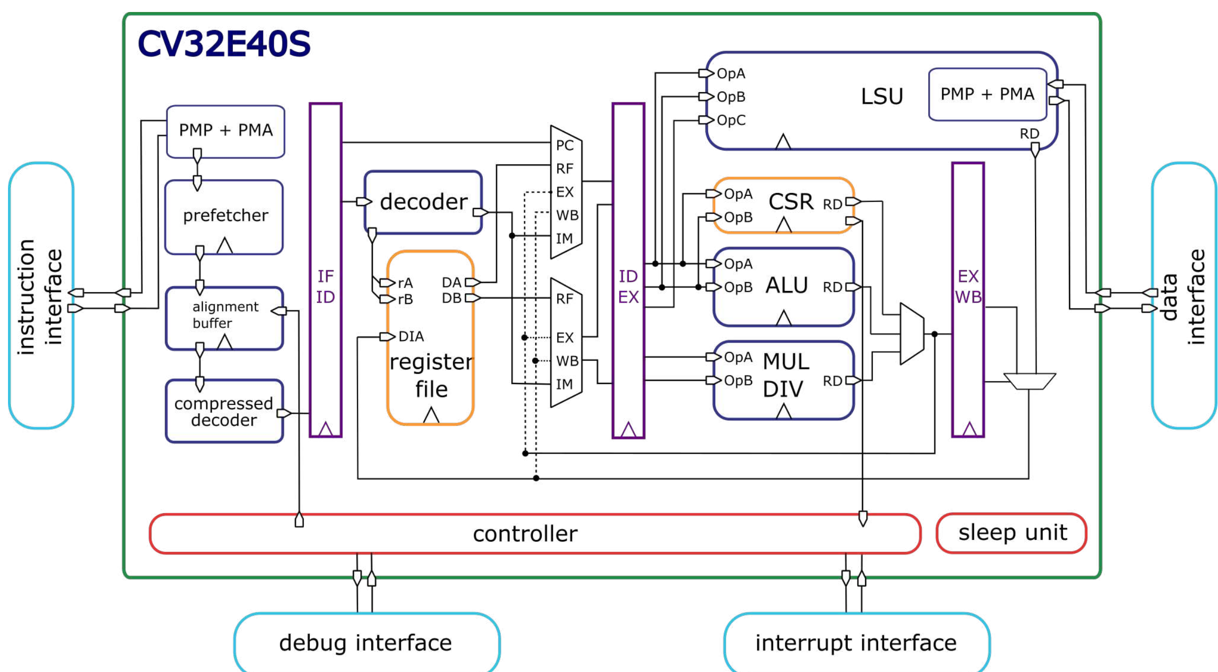


Figura 17 – Esquemático do núcleo CV32E40S.

O microprocessador CV32E40S foi projetado para aplicativos de segurança e oferece modo de máquina e modo de usuário, Physical Memory Protection (PMP) aprimorado e vários recursos anti-sabotagem.

Esse kernel usa uma extensão personalizada chamada *Xsecure*, que inclui os seguintes recursos de segurança:

- **Data independent timing**: se o bit dataindtiming for definido no registro Control and Status Register (CSR) cpuctrl, o tempo de execução de todas as instruções será independente dos dados de entrada para dificultar a extração de informações por um observador externo, observando o consumo de energia ou explorando a sincronização *side channel*.
  Quando o bit dataindtiming é definido, as instruções DIV, DIVU, REM e REMU têm uma latência fixa e as ramificações também têm uma latência fixa, independentemente de serem usadas ou não.
- **Dummy instruction insertion**: se o bit rnddummy for definido no registro cpuctrl CSR, essa contramedida inserirá instruções fictícias em intervalos aleatórios no *pipeline* de execução. As instruções fictícias não têm impacto funcional sobre o estado do processador, mas acrescentam distúrbios ao código executado que são difíceis de prever em termos de tempo e energia. Essas interrupções tornam mais difícil para um atacante deduzir o que está sendo executado e mais difícil executar ataques de injeção de falhas no momento exato.
  A frequência das instruções injetadas pode ser definida usando os bits rnddummyfreq no CSR cpuctrl.

rnddummyfreq	Intervalo
0000	Instruções fictícias a cada 1 a 04 instruções reais
0001	Instruções fictícias a cada 1 a 08 instruções reais
0011	Instruções fictícias a cada 1 a 16 instruções reais
0111	Instruções fictícias a cada 1 a 32 instruções reais
1111	Instruções fictícias a cada 1 a 64 instruções reais

Tabela 3. Intervalos para os parâmetros rnddummyfreq.

A frequência de inserção da instrução fictícia é randomizada usando um registrador de deslocamento chamado LFSR0 - Linear Feedback Shift Register (LFSR). A instrução fictícia em si também é randomizada com base no LFSR0 e é limitada às instruções add, mul, and e bltu. Os dados de origem das instruções fictícias são obtidos do LFSR (LFSR1 e LFSR2) e não do arquivo de registro.

A semente inicial e a permutação de saída do LFSRs podem ser definidas usando os seguintes parâmetros do nível superior do CV32E40S:

- LFSR0_CFG para LFSR0;

  - LFSR1_CFG para LFSR1;
  - LFSR2_CFG para LFSR2.
- **Random instruction for hint**: se o bit rndhint for definido no registro CSR cpuctrl, a instrução c.ssli com rd=x0, nzimm!=0 no uso do RVC personalizado será substituída por uma instrução aleatória. A instrução aleatória não tem impacto funcional sobre o estado do processador (ou seja, é funcionalmente equivalente a um nop, mas pode causar uma diferença no número de ciclos, na busca de instruções e no comportamento energético). A instrução aleatória é gerada aleatoriamente a partir do LFSR0 e é limitada às instruções add, mul, and e bltu. Os dados de origem da instrução aleatória vêm do LFSR (LFSR1 e LFSR2) em vez do arquivo de registro;
- ***Hardened PC***: se o bit pcharden for definido no registro CSR cpuctrl, durante a execução sequencial, o PC no estágio IF é reforçado verificando-se se ele tem o valor correto em relação ao estágio ID, com um deslocamento determinado pelo estado comprimido/não comprimido da instrução em ID.
  Além disso, o PC da etapa IF é verificado quanto à exatidão no caso de uma possível execução não sequencial devido à transferência de instruções de controle. Para saltos (incluindo modo de máquina - mret) e ramificações, isso é feito recalculando o destino do PC e a decisão de ramificação (gerando um ciclo extra para ramificações não realizadas). Qualquer erro na verificação do PC correto ou na decisão de ramificação/pulo resultará em um impulso no sinal alert_major_o.
- **Interface Integrity**: O processador realiza verificações de integridade por meio de sinais de paridade e soma de verificação nas interfaces de barramento Open Bus Interface (OBI) (OpenHW Group 2024) para transações de instruções e dados. O processador gera sinais de paridade e somas de verificação ímpares, enquanto o ambiente, ou seja, um dispositivo fora do microprocessador, como a memória, deve fornecer sinais de paridade e somas de verificação correspondentes.
  Há dois tipos de verificação de integridade:
  - **Unrelated checks**: Essas verificações são realizadas independentemente da execução de uma instrução específica, observando apenas as interfaces de barramento. Os erros de paridade e de soma de verificação detectados acionam um alerta por meio do pino alert_major_o.
  - **Associated checks**: essas verificações são realizadas durante a execução de instru-

ções específicas, como fetch, load ou store. Erros de paridade ou *checksum* nessas interfaces podem resultar em exceções específicas ou interrupções não mascaráveis com códigos de exceção específicos, além de acionar um alerta.

As verificações de *checksum* só são realizadas se estiverem ativadas globalmente (o bit integrity no registro CSR cpuctrl) e localmente (via *integrity* Physical Memory Attribution (PMA), enquanto as verificações de paridade estão sempre ativadas.

- **Bus protocol hardening**: o protocolo OBI é usado como protocolo para a interface de instruções e a interface de dados do CV32E40S. Para os sinais de contato (*req*, *gnt*, *rvalid*), a principal violação do protocolo é receber uma resposta quando não há nenhuma transação correspondente em andamento. Se ocorrer um erro, o microprocessador acionará o sinal alert_major_o;
- **Register file Error Correction Code (ECC)**: a verificação ECC (*Correcting Code*) é adicionada a todas as leituras do arquivo de registro, em que há uma soma de verificação para cada palavra de registro. Os erros de 1 bit ocorrem quando um único bit em um item de dados é alterado incorretamente, enquanto os erros de 2 bits ocorrem quando dois bits são alterados. Todos esses erros serão detectados pelo sistema ECC. Isso pode ser útil para detectar ataques de injeção de falhas, pois o arquivo de registro cobre uma área relativamente grande do CV32E40S. Não é feita nenhuma tentativa de corrigir os erros detectados, mas um alerta importante é acionado se um erro for detectado, permitindo que o sistema tome providências.
- **Hardened CSRs**: para registros de CSR essenciais, é adicionada uma segunda cópia do registro, que armazena uma versão completa dos dados principais do CSR. Uma verificação constante é feita para garantir que as duas cópias sejam consistentes, e um alerta importante é acionado se não forem.

Quando o sistema detecta um problema de segurança, não é feita nenhuma tentativa de corrigi-lo. No entanto, o sistema usa dois sinais para indicar um problema de segurança para que o usuário possa tomar medidas adicionais. Os sinais de aviso são:

- ***alert_major_o***: indica problemas críticos de segurança dos quais o kernel não pode se recuperar;
- ***alert_minor_o***: indica problemas potenciais de segurança que podem ser monitorados por um sistema durante um período de tempo.

3.2.1 Preparação das contramedidas no código

Para determinar quais contramedidas são implementadas, o registro de controle e status CSR correspondente à extensão *Xsecure* deve ser modificado. De acordo com o manual do usuário do CV32E40S (OpenHW Group 2024), o registro *CPU Control* é responsável por ativar e desativar as contramedidas. As contramedidas que podem ser modificadas são:

- Data independent timing (bit 0);
- Dummy instruction insertion (bit 1);
- Random instruction for hint (bit 2);
- Hardened PC (bit 3);
- Interface Integrity (bit 4).

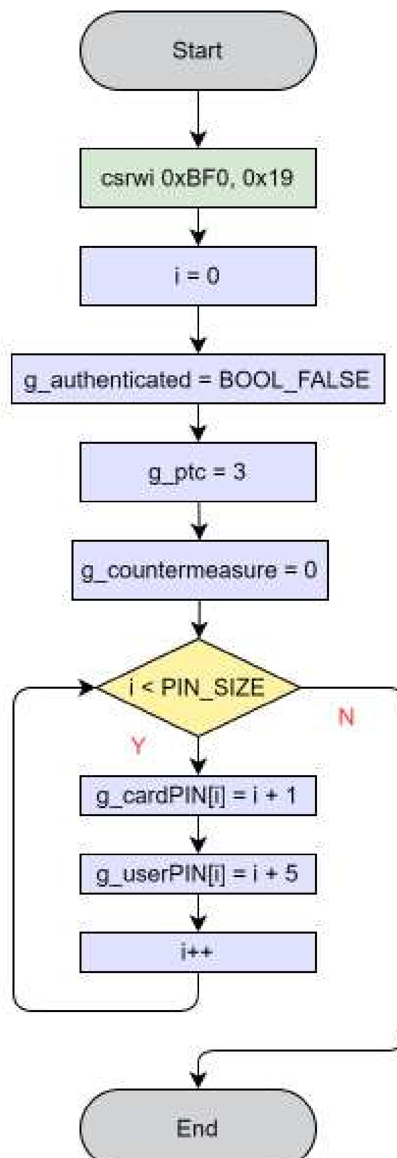


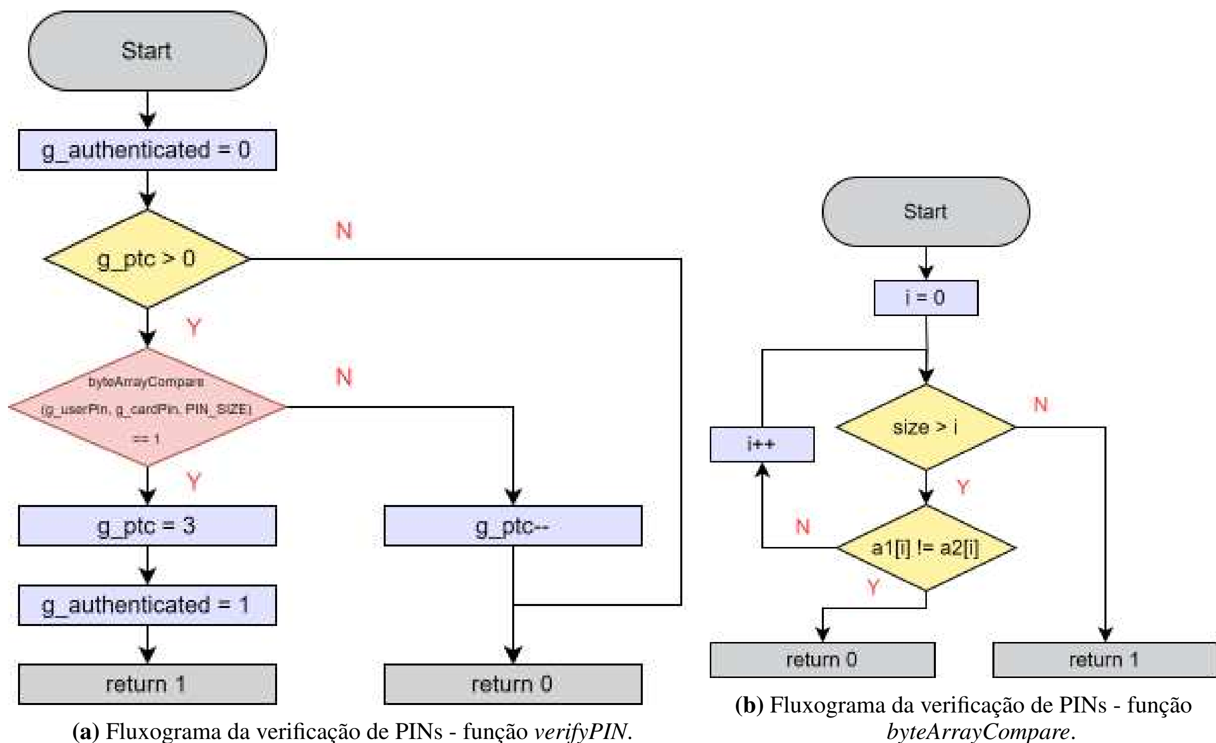
Figura 18 – Fluxograma da inicialização do PIN com o comando para inicializar a contramedida desejada.

Seu endereço é 0xBF0 e seu valor padrão é 0x0000_0019, o que significa que as contramedidas ativadas por padrão são *Data independent timing*, *Hardned PC* e *Interface Integrity*.

A fim de controlar as contramedidas incorporadas, foi adicionado um comando à função *initialize* (consulte o apêndice A), conforme mostrado na Figura 18. Como resultado, é possível determinar quais das contramedidas listadas acima serão ativadas e/ou desativadas.

3.2.2 Preparação do código a ser executado

Como o microprocessador é destinado a aplicativos de segurança, foi escolhido a versão zero da coleção FISSC (consulte o apêndice A) para avaliar a robustez do CV32E40S. A versão zero não contém mecanismo de defesa, ao contrário da versão sete, o que significa que as soluções avaliadas são exclusivamente baseadas em *hardware*, como se pode ver na Figura 19. Ela realiza uma comparação entre o PIN do usuário e o PIN armazenado no sistema e, se detectar uma diferença inicial entre os PINs, encerrará automaticamente sua operação e negará a autorização ao usuário.



(a) Fluxograma da verificação de PINs - função *verifyPIN*.

(b) Fluxograma da verificação de PINs - função *byteArrayCompare*.

Figura 19 – Fluxogramas do processo de verificação de PINs da versão zero.

3.2.3 Avaliação da robustez do CV32E40S

O modelo de ataque usado é semelhante ao usado para o CV32E40P: ele visa injetar uma única falha na lógica sequencial do caminho de controle do microprocessador durante o intervalo de tempo em que ocorre a verificação do PIN. Foi selecionado todos os fios de saída dos registros com larguras entre 1 e 8 bits para aplicar uma injeção de efeito diferente do valor correto; no entanto, a injeção será ativada durante todo o período da função *VerifyPIN*, pois esse código não tem contramedidas de *software*. Dessa forma, o modelo de ataque visava 95 registros. Observe que foi removido os sinais de alerta e os registros CSR.

```
select t:*ff* %co t:*ff* %d s:1:7 %i
select -del
    cv32e40s_yosys_wrapper/cv32e40s_core_i.alert_i.alert_minor_o
    cv32e40s_yosys_wrapper/cv32e40s_core_i.alert_i.alert_major_o
    cv32e40s_yosys_wrapper/cv32e40s_core_i.cs_registers_i*
fault_rtlil -effect diff -cnt 1
```

Figura 20 – Comandos Yosys utilizados para injetar falhas no CV32E40S.

Como esse microprocessador usa dois sinais para indicar problemas de segurança, o objetivo do atacante será burlar os mecanismos de defesa padrão do microprocessador e aqueles escolhidos pelo usuário, que são definidos na função de inicialização:

$\varphi_S :=$ (authenticated $\wedge \neg$ hardware alert)

Assim, a avaliação da robustez diz respeito a todas as contramedidas de *hardware*.

Para fazer isso, é necessário encontrar os locais de memória onde os dados variáveis estão armazenados. Repetindo as etapas aplicadas ao CV32E40P, foi desmontado o código para descobrir os endereços de cada variável (consulte o apêndice A para obter o arquivo desmontado). A Tabela 4 mostra os respectivos endereços de cada variável:

Variável	Endereço hexadecimal	Endereço binário	Valor armazenado
g_cardPin[0]	0x148	1_0100_1000	1
g_cardPin[1]	0x149	1_0100_1001	2
g_cardPin[2]	0x14A	1_0100_1010	3
g_userPin[0]	0x14C	1_0100_1100	5
g_userPin[1]	0x14D	1_0100_1101	6
g_userPin[2]	0x14E	1_0100_1110	7
g_authenticated	0x151	1_0101_0001	0 (Falso)

Tabela 4. Endereços e valores de cada variável usada no código *verifypin_v0.c*.

```
assign PIN1 = mem['b101001000] != mem['b101001100];
assign PIN2 = mem['b101001001] != mem['b101001101];
assign PIN3 = mem['b101001010] != mem['b101001110];
assume property(PIN1);
assume property(PIN2);
assume property(PIN3);
assign unauthenticated = (mem['b101010001] != 8'h1);
assert property(unauthenticated);
assume property(~alert_minor_o);
assume property(~alert_major_o);
```

Figura 21 – Propriedades estabelecidas para o CV32E40S verificar.

As configurações de contramedidas escolhidas foram as seguintes: nenhuma contramedida; a configuração padrão do microprocessador (*Data Independent Timing*, *Hardened PC* e *Interface Integrity*); a configuração para a contramedida *Dummy Instructions Insertion*; a configuração para a contramedida *Random Instruction for Hint*; bem como a configuração que habilita todas as contramedidas, em que a frequência de inserção de instruções fictícias é de 1 a 4.

3.2.4 Resultados obtidos

Foi seguido o mesmo procedimento ilustrado na figura 15: simular até o ponto de interesse, ou seja, antes da função *verifyPIN*, substituir os valores PIN por valores simbólicos, injetar os multiplexadores no circuito, gerar os arquivos do sistema de transição e, em seguida, proceder à verificação. A Tabela 5 abaixo mostra os tempos de simulação para cada configuração executada no Questasim: o ciclo do relógio, o tempo em que a função *verifyPIN* será chamada e o tempo total de execução de todas as instruções desde que o processador foi iniciado.

Contramedidas ativadas	Ciclo de *clock* (ns)	Início da função verifyPIN		Duração total de execução	
		Tempo (ns)	Ciclo	Tempo (ns)	Ciclo
Nenhuma	10	775	77,5	1335	133,5
Padrão	10	795	79,5	1435	143,5
rnddummy	10	915	91,5	1645	164,5
rndhint	10	775	77,5	1335	133,5
Todas	10	1025	102,5	1835	183,5

Tabela 5. Tempo de simulação para cada configuração executada no Questasim.

Com base na Tabela 5, foi determinado o tempo de simulação para cada configuração no Yosys e também o tempo de verificação k = 100 para cobrir todo o período de execução, desde o início da função *verifyPIN* até o final da execução. Observe que, depois que todas as instruções forem executadas, o microprocessador executará a função *exit* (consulte o apêndice A) que foi adicionada por meio do Board Support Package (BSP) para concluir sua execução. A Tabela 6 exibe os resultados obtidos na verificação para cada configuração:

Contramedidas ativadas	Resultado BMC	K detectado	Sinal empregado	Tempo da verificação formal
Nenhuma	Alcançável	23	n_flush_q	01h04min
Padrão	Alcançável	24	n_flush_q	01h13min
rnddummy	Alcançável	41	is_clic_ptr_o	11h42min
rndhint	Não Alcançável	X	X	11h22min
Todas	Não Alcançável	X	X	10h18min

Tabela 6. Resultados da verificação formal.

Os resultados indicam que as contramedidas padrão (*dataindtiming*, *pcharden* e *integrity*) e a contramedida *rnddummy* não conseguem gerenciar com eficácia uma injeção de falha no sistema. De fato, a verificação formal da BMC detectou que uma das propriedades definidas não foi respeitada, permitindo que o atacante obtivesse o estado autenticado sem acionar os alertas de *hardware*. Observe que a verificação é interrompida assim que uma exploração inicial é encontrada, o que significa que outras falhas potencialmente exploráveis podem não ser detectadas. Portanto, é necessário reiniciar a simulação para identificá-las.

Os sinais usados para os cenários sem contramedidas, a configuração padrão e com a contramedida *rnddummy* vêm do módulo *prefetch_ctrl* do estágio IF. Esses sinais funcionam na sincronização dos PCs entre os estágios IF e ID. Ao explorar a dessincronização entre esses estágios, é possível ignorar determinadas instruções, permitindo que o atacante pule as instruções de comparação antes de atribuir o valor verdadeiro à variável *g_authenticated*.

Por outro lado, para a contramedida *rndhint* e para a configuração que ativa todas as contramedidas, os resultados mostram que o objetivo do atacante não pode ser alcançado. Isso significa que o microprocessador conseguiu lidar com as injeções de falhas sem permitir o acesso não autorizado. Um possível motivo para esse sucesso é que, embora uma injeção tenha sido aplicada ao módulo *prefetch_ctrl*, as instruções ignoradas foram aquelas geradas pela contramedida rndhint, o que impediu que as instruções reais fossem suprimidas.

4 CONCLUSÃO

Ao longo deste trabalho, foi possível aprofundar conhecimentos em áreas fundamentais, como arquitetura de processadores, programação de baixo nível, *hardware* e segurança de *hardware*. Inicialmente, foi dominado o uso de ambientes de simulação RTL, a injeção de falhas na simulação e a geração de pequenos programas para microprocessadores RISC-V de 32 bits. Com base nos estudos de caso e nos artigos FDTC e FMCAD, foi possível utilizar a ferramenta μArchiFI, reproduzir os estudos de caso e, em seguida, realizar trabalhos com o microprocessador CV32E40S.

Usando o código verifypin_v0.c para realizar uma análise de robustez do *hardware*, os resultados obtidos são diferentes dependendo da configuração:

- Para os cenários em que as contramedidas estão desativadas e as contramedidas padrão estão ativadas, o solucionador encontra uma vulnerabilidade no sinal n_flush_q no módulo prefetch_ctrl, que é conhecido e listado na Tabela II do artigo do FDTC para o estudo realizado em sua versão P. Para o cenário de contramedida *Dummy instructions insertion*, o solucionador encontra uma vulnerabilidade no mesmo módulo. De acordo com o artigo da FDTC, esses sinais são usados para sincronizar o PC entre as etapas IF e ID, e a manipulação desse procedimento pode levar a saltos de instruções, que podem incluir instruções de decisão para atribuir valores a variáveis.
- No entanto, para o cenário de contramedida *Random instruction for hint*, o solucionador não encontrou vulnerabilidades quando verificado, o que não era esperado a primeiro momento porque a função verifypin_v0.c não apresenta instruções personalizadas. Logo, seria interessante realizar uma análise mais aprofundada dessa contramedida.

4.1 Trabalhos futuros

Como continuação deste trabalho, uma etapa inicial pode consistir na avaliação detalhada dos resultados obtidos, com ênfase nas contramedidas *Dummy instructions insertion* e *Random instruction for hint*. Embora o solucionador tenha identificado uma vulnerabilidade na primeira, nenhuma foi detectada na segunda, apesar de ambas utilizarem o mesmo LFSR para inserção de instruções aleatórias. Além disso, a análise da diferença de tempo entre as verificações realizadas pode fornecer informações relevantes sobre a eficácia temporal das contramedidas.

Após a identificação inicial de vulnerabilidades, uma reexecução das verificações, com atraso de um ciclo após a detecção da falha, pode revelar vulnerabilidades adicionais latentes no tempo de execução. Esse tipo de análise permitiria uma compreensão mais aprofundada do comportamento do sistema frente a ataques, abrangendo falhas imediatas e tardias.

Por fim, uma comparação entre os resultados obtidos para o pacote CV32E40P associado ao código verifypin_v7.c e aqueles relacionados ao CV32E40S pode revelar semelhanças e diferenças na resposta das duas arquiteturas a ataques e contramedidas semelhantes.

4.2 Dificuldades encontradas

Atualmente, a página oficial do OpenHW Group indica que o microprocessador CV32E40S ainda se encontra em desenvolvimento, o que implica na ausência de cobertura completa. De fato, durante a utilização deste microprocessador, foram identificados diversos erros que impossibilitaram a realização adequada das análises.

O compilador destinado à conversão de arquivos .c para .hex no CV32E40S encontra-se incompleto e não oferece suporte a todos os conjuntos de instruções. Em virtude disso, foi necessário utilizar uma versão anterior do compilador, mais estável, porém também limitada em relação ao suporte completo de instruções. Diante dessa limitação, optou-se por utilizar o conjunto de instruções I-M-C-Zicsr, essencial para o funcionamento adequado tanto do microprocessador quanto dos códigos verifypins.

Embora o microprocessador possa ser simulado no ambiente de programação CORE-V-VERIF, com o apoio das bibliotecas UVM, o conjunto microprocessador + memória (conforme ilustrado na Figura 11) não era sintetizável, pois a simulação realizava uma análise comportamental do core S, sem empregar uma memória dedicada. Ressalta-se que os microprocessadores, tanto o core P quanto o core S, não possuem o estágio de memória, o que exigiu a utilização de uma memória externa para suprir essa necessidade.

Contudo, apesar da presença de uma memória no repositório do CV32E40S, esta não funcionava corretamente nos testes. Em particular, durante o acesso à memória por meio das instruções de armazenamento e carregamento, o microprocessador interrompia sua execução. Ao contatar o OpenHW Group sobre o problema, foi informado que não havia uma memória dedicada ao CV32E40S. Diante disso, e considerando que a memória do core P funcionava corretamente, esta foi adaptada ao core S, com base tanto na documentação OBI quanto na documentação do microprocessador, especialmente no que diz respeito à contramedida *interface*

integrity, responsável por gerenciar os acessos à memória.

Além disso, foi necessário adaptar o BSP a fim de reduzir a quantidade de instruções geradas pela versão original, bem como para compatibilizá-lo com a nova memória. A versão original gerava milhares de instruções adicionais às bibliotecas UVM, enquanto a versão modificada passou a gerar apenas as instruções essenciais à execução dos códigos.

Após a implementação das instruções descritas na documentação, a memória foi testada por meio de simulações RTL, com o objetivo de garantir seu funcionamento correto e sua capacidade de síntese. Durante a simulação, foi possível observar os sinais de acesso à memória, bem como a execução completa das instruções (principalmente as de armazenamento e carregamento), o que confirmou o funcionamento esperado da memória, sem a ativação da contramedida de integridade.

Em seguida, foi necessário converter os arquivos em SystemVerilog para Verilog, uma vez que o fluxo de trabalho da ferramenta μArchiFI aceita apenas arquivos escritos em Verilog em sua versão gratuita. Para isso, foi utilizada a ferramenta sv2v. No entanto, essa conversão resultou em diversos erros relacionados ao uso de funções de aleatoriedade (por exemplo, $urandom), as quais não possuem equivalente direto na linguagem Verilog, além de funções pertencentes à biblioteca UVM. Essas funções, empregadas para simular latências nos acessos à memória, não são sintetizáveis e são utilizadas exclusivamente em simulações. Como solução, foram inseridos blocos condicionais *if-else* no código para ignorar essas estruturas durante o processo de conversão.

Todo esse processo demandou um tempo considerável, configurando-se como a etapa mais longa deste trabalho. Para assegurar o correto funcionamento do microprocessador, foi necessário identificar e analisar os problemas encontrados, o que envolveu consultas detalhadas a documentos técnicos, bem como o contato direto com os projetistas responsáveis pelo desenvolvimento dos microprocessadores.

REFERÊNCIAS

BOOLECTOR. **Boolector: SMT solver for bit-vectors and arrays**. 2024. Disponível em: <https://github.com/Boolector/boolector/>.

CENTAUR, S. **Pono: SMT-Based Model Checker**. 2024. Disponível em: <https://github.com/stanford-centaur/pono>.

DUREUIL, L.; PETIOT, G.; POTET, M.; LE, T.; CROHEN, A.; CHOUDENS, P. de. FISSC: A fault injection and simulation secure collection. In: **Computer Safety, Reliability, and Security - 35th International Conference, SAFECOMP 2016, Trondheim, Norway, September 21-23, 2016, Proceedings**. [S.l.: s.n.], 2016. p. 3–11.

Embecosm. **Toolchain para CORE-V**. 2024. Disponível em: <https://www.embecosm.com/resources/tool-chain-downloads/#corev>.

GTKWave. **Site oficial do GTKWave**. 2024. Disponível em: <https://gtkwave.sourceforge.net/>.

HEYDEMANN, K. **Attaques par injection de fautes et protections logicielles**. 2024. Disponível em: <https://www.college-de-france.fr/media/xavier-leroy/UPL6178057278649719790_Heydemann.pdf>.

OpenHW Group. **Depósito do CV32E40P**. 2024. Disponível em: <https://github.com/openhwgroup/cv32e40p>.

OpenHW Group. **Depósito do CV32E40S**. 2024. Disponível em: <https://github.com/openhwgroup/cv32e40s>.

OpenHW Group. **Documentação do Open Bus Interface protocol**. 2024. Disponível em: <https://github.com/openhwgroup/obi/blob/072d9173c1f2d79471d6f2a10eae59ee387d4c6f/OBI-v1.6.0.pdf>.

OpenHW Group. **Manual do usuário do CORE-V-VERIF**. 2024. Disponível em: <https://docs.openhwgroup.org/projects/core-v-verif/en/latest/intro.html>.

OpenHW Group. **Manual do usuário do CV32E40P**. 2024. Disponível em: <https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/intro.html>.

OpenHW Group. **Manual do usuário do CV32E40S**. 2024. Disponível em: <https://docs.openhwgroup.org/projects/cv32e40s-user-manual/en/latest/>.

Podman. **Site oficial do Podman**. 2024. Disponível em: <https://podman.io>.

SNOW, Z. **Repositório do sv2v**. 2024. Disponível em: <https://github.com/zachjs/sv2v>.

TOLLEC, S.; ASAVOAE, M.; COUROUSSé, D.; HEYDEMANN, K.; JAN, M. Exploration of fault effects on formal risc-v microarchitecture models. In: **Workshop on Fault Detection and Tolerance in Cryptography (FDTC)**. [S.l.: s.n.], 2022. p. 73–83. Evento virtual, Itália.

TOLLEC, S.; ASAVOAE, M.; COUROUSSé, D.; HEYDEMANN, K.; JAN, M. μarchifi: Formal modeling and verification strategies for microarchitetural fault injections. In: **Formal Methods in Computer-Aided Design (FMCAD)**. [S.l.: s.n.], 2023. p. 101–109. Ames, IO, Estados Unidos.

TOLLEC, S.; HADžIć, V.; NASAHL, P.; ASAVOAE, M.; BLOEM, R.; COUROUSSé, D.; HEYDEMANN, K.; JAN, M.; MANGARD, S. Fault-resistant partitioning of secure cpus for system co-verification against faults. **IACR Transactions on Cryptographic Hardware and Embedded Systems**, 2024.

Verilator. **Site oficial do Verilator**. 2024. Disponível em: <https://www.veripool.org/verilator/>.

YosysHQ. **Repositório do Yosys**. 2024. Disponível em: <https://github.com/YosysHQ/yosys>.

APÊNDICE A – CÓDIGOS UTILIZADOS NESTE TRABALHO

Arquivo verifypin_v7.c

Código A1 - Função main() do arquivo verifypin_v7.c

```
// This file is part of FISSC.
//
// you can redistribute it and/or modify it under the terms of the GNU
// Lesser General Public License as published by the Free Software
// Foundation, version 3.0.
//
// It is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// See the GNU Lesser General Public License version 3.0
// for more details (enclosed in the file LICENSE).

/*$
  @name = VerifyPIN_6_HB+FTL+INL+DPTC+DT
  @authors = Lionel Riviere, SERTIF Consortium
  @feature = verifyPIN
  @target = satisfied oracle
  @countermeasure = Hardened booleans, inlined calls, Decrement PTC first, Double test, Fixed time loop
  @difficulties = none
  @purpose = N/A
*/

#include <stddef.h>
```

```
26
27 BOOL verifyPIN_6(void);
28 extern BOOL g_authenticated;
29
30 int main()
31 {
32     initialize();
33     verifyPIN_6();
34     return 0;
35 }
```

Código A2 - Função initialize() do arquivo verifypin_v7.c

```
1  // Macro definitions
2  typedef signed char SBYTE;
3  typedef unsigned char UBYTE;
4  typedef unsigned char BOOL;
5  typedef unsigned long ULONG;
6
7  #define BOOL_TRUE 0xAA
8  #define BOOL_FALSE 0x55
9
10 #define INITIAL_VALUE 0x2a
11 #define PIN_SIZE 3
12
13 // Global variables definition
14 BOOL g_authenticated;
15 SBYTE g_ptc;
16 UBYTE g_countermeasure;
17 UBYTE g_userPin[PIN_SIZE];
18 UBYTE g_cardPin[PIN_SIZE];
19
20 void initialize()
21 {
```

```
22    // Local variables
23    int i;
24    // Global variables initialization
25    g_authenticated = BOOL_FALSE;
26    g_ptc = 3;
27    g_countermeasure = 0;
28    // Card PIN = 1 2 3 4 5...
29    // User PIN = 5 6 7 8 9...
30    for (i = 0; i < PIN_SIZE; ++i) {
31        g_cardPin[i] = i+1;
32        g_userPin[i] = i+5;
33    }
34 }
```

Código A3 - Função verifyPIN_6() do arquivo verifypin_v7.c

```
1  extern SBYTE g_ptc;
2  extern BOOL g_authenticated;
3  extern UBYTE g_userPin[PIN_SIZE], g_cardPin[PIN_SIZE];
4  #if defined INLINE && defined PTC
5  inline BOOL verifyPIN_6() __attribute__((always_inline))
6  #else
7  BOOL verifyPIN_6()
8  #endif
9  {
10   int i;
11   BOOL status, diff;
12   g_authenticated = BOOL_FALSE;
13   if(g_ptc >= 0) {
14     g_ptc--;
15     status = BOOL_FALSE;
16     diff = BOOL_FALSE;
17     for(i = 0; i < PIN_SIZE; i++) {
18       if(g_userPin[i] != g_cardPin[i]) {
```

```
19          diff = BOOL_TRUE;
20        }
21      }
22      if(i != PIN_SIZE) {
23        countermeasure();
24      }
25      if (diff == BOOL_FALSE) {
26        if(BOOL_FALSE == diff) {
27          status = BOOL_TRUE;
28        } else {
29          countermeasure();
30        }
31      } else {
32        status = BOOL_FALSE;
33      }
34      if(status == BOOL_TRUE) {
35        if(BOOL_TRUE == status) {
36          g_ptc = 3;
37          g_authenticated = BOOL_TRUE;
38          return BOOL_TRUE;
39        } else {
40          countermeasure();
41        }
42      }
43    }
44    return BOOL_FALSE;
45  }
46  void countermeasure()
47  {
48      g_countermeasure = 1;
49  }
```

Código A4 - Instruções da função initialize() do arquivo verifypin_v7.c

```
000000ca <initialize>:
  ca: 05500713            addi    a4,zero,85
  ce: 12e00ea3            sb      a4,317(zero) # 13d <
     g_authenticated>
  d2: 470d                c.li    a4,3
  d4: 12e00e23            sb      a4,316(zero) # 13c <g_ptc>
  d8: 12000da3            sb      zero,315(zero) # 13b <
     g_countermeasure>
  dc: 4781                c.li    a5,0
  de: a00d                c.j     100 <initialize+0x36>
  e0: 0ff7f693            andi    a3,a5,255
  e4: 00168613            addi    a2,a3,1
  e8: 13400713            addi    a4,zero,308
  ec: 973e                c.add   a4,a5
  ee: 00c70023            sb      a2,0(a4)
  f2: 0695                c.addi  a3,5
  f4: 13800713            addi    a4,zero,312
  f8: 973e                c.add   a4,a5
  fa: 00d70023            sb      a3,0(a4)
  fe: 0785                c.addi  a5,1
 100: 4709                c.li    a4,2
 102: fcf75fe3            bge     a4,a5,e0 <initialize+0x16>
 106: 8082                c.jr    ra
```

Arquivo verifypin_v0.c

Código A5 - Função initialize() do arquivo verifypin_v0

```c
// Macro definitions
typedef signed char SBYTE;
typedef unsigned char UBYTE;
typedef unsigned char BOOL;
typedef unsigned long ULONG;

#define BOOL_TRUE 0xAA
#define BOOL_FALSE 0x55

#define INITIAL_VALUE 0x2a
#define PIN_SIZE 3

// Global variables definition
BOOL g_authenticated;
SBYTE g_ptc;
UBYTE g_countermeasure;
UBYTE g_userPin[PIN_SIZE];
UBYTE g_cardPin[PIN_SIZE];

void initialize()
{
   // Set the cpuctrl register
   __asm__ volatile("csrwi 0xBF0, 0x19");
   // local variables
   int i;
   // global variables initialization
   g_authenticated = 0;
   g_ptc = 3;
   g_countermeasure = 0;
   // card PIN = 1 2 3 4 5...
   // user PIN = 5 6 7 8 9...
```

```
32     for (i = 0; i < PIN_SIZE; ++i) {
33         g_cardPin[i] = i+1;
34         g_userPin[i] = i+5;
35     }
36 }
```

Código A6 - Instruções da função initialize() do arquivo verifypin_v0.c

```
1  00000038 <initialize>:
2    38:    bf0cd073            csrrwi  x0,0xbf0,25
3    3c:    140008a3            sb   x0,337(x0) # 151 <
       g_authenticated>
4    40:    470d                c.li    x14,3
5    42:    14e00823            sb   x14,336(x0) # 150 <g_ptc>
6    46:    140007a3            sb   x0,335(x0) # 14f <
       g_countermeasure>
7    4a:    4781                c.li    x15,0
8    4c:    a00d                c.j 6e <initialize+0x36>
9    4e:    0ff7f693            andi    x13,x15,255
10   52:    00168613            addi    x12,x13,1
11   56:    14800713            addi    x14,x0,328
12   5a:    973e                c.add   x14,x15
13   5c:    00c70023            sb   x12,0(x14)
14   60:    0695                c.addi  x13,5
15   62:    14c00713            addi    x14,x0,332
16   66:    973e                c.add   x14,x15
17   68:    00d70023            s    x13,0(x14)
18   6c:    0785                c.addi  x15,1
19   6e:    4709                c.li    x14,2
20   70:    fcf75fe3            bge x14,x15,4e <initialize+0x16>
21   74:    8082                c.jr    x1
```

Código A7 - Função verifyPIN() do arquivo verifypin_v0.c

```
extern SBYTE g_ptc;
extern BOOL g_authenticated;
extern UBYTE g_userPin[PIN_SIZE];
extern UBYTE g_cardPin[PIN_SIZE];

#ifdef INLINE
__attribute__((always_inline)) inline BOOL byteArrayCompare(
    UBYTE* a1, UBYTE* a2, UBYTE size)
#else
BOOL byteArrayCompare(UBYTE* a1, UBYTE* a2, UBYTE size)
#endif
{
    int i;
    for(i = 0; i < size; i++) {
        if(a1[i] != a2[i]) {
            return 0;
        }
    }
    return 1;
}

BOOL verifyPIN() {
    g_authenticated = 0;

    if(g_ptc > 0) {
        if(byteArrayCompare(g_userPin, g_cardPin, PIN_SIZE) ==
            1) {
            g_ptc = 3;
            g_authenticated = 1; // Authentication();
            return 1;
        } else {
            g_ptc--;
            return 0;
```

```
32          }
33      }
34
35      return 0;
36  }
```

Código A8 - Função exit() do arquivo verifypin_v0.c

```
1  #define EXIT_REG   0x20000004
2  void exit(int exit_status) {
3      *(volatile int *)EXIT_REG = exit_status;
4      __asm__ volatile("wfi");
5      while(1){};
6  }
```