DinSecUEFI: Testes Dinâmicos em *Pipeline* CI para *Firmware* baseado na especificação UEFI

Gelson José de A. Filho[1], Juliana de Santi[1], Newton Carlos Will[1], Rafael R. Machado[2], Andréia Leles[2]

[1]Programa de Pós-graduação em Computação Aplicada (PPGCA) - Universidade Tecnológica Federal do Paraná (UTFPR) - Curitiba, PR - Brasil

[2]Centro Universitário FACENS - Sorocaba, SP – Brasil

gelsonfilho@alunos.utfpr.edu.br,{jsanti,will}@utfpr.edu.br

{rafael.machado,andreia.leles}@facens.br

Abstract. *Firmware security is critical due to its elevated privileges and the absence of defenses typical of operating systems (OSs). While OSs have multiple layers of protection, firmware remains more vulnerable and attractive to attacks. Following DevSecOps principles, this work proposes a pipeline that automates dynamic testing in UEFI modules. The methodology uses binary instrumentation and fuzzing executed in containers for scalable dynamic analysis. In experiments with a real BIOS, the pipeline identified failures in more than 5% of the analyzed modules. The results indicate that the approach reduces operational costs and strengthens the secure development cycle of UEFI-based firmware.*

Resumo. *A segurança de firmware é crítica devido a seus privilégios elevados e à ausência de defesas típicas de sistemas operacionais (SOs). Enquanto os SOs contam com múltiplas camadas de proteção, o firmware permanece mais vulnerável e atrativo para ataques. Seguindo os princípios do DevSecOps, este trabalho propõe um pipeline que automatiza testes dinâmicos em módulos UEFI. A metodologia utiliza instrumentação binária e fuzzing executado em contêineres para análise dinâmica e escalável. Em experimentos com uma BIOS real, o pipeline identificou falhas em mais de 5% dos módulos analisados. Os resultados indicam que a abordagem reduz custos operacionais e fortalece o ciclo de desenvolvimento seguro de firmware baseado na UEFI.*

1. Introdução

Ataques ao *firmware* têm se tornado atraentes devido à carência relativa de tecnologias de segurança eficazes nesse âmbito, geralmente menos protegido que os sistemas operacionais tradicionais. Embora existam recursos como *Secure Boot*, *BIOS Password* e *Revocation Lists*, tais medidas isoladas não mitigam completamente os desafios específicos desse contexto — como a possibilidade de persistência em nível elevado de privilégio, a furtividade dos ataques e o impacto significativo na reputação das empresas quando vulnerabilidades são exploradas [Yao and Zimmer 2020].

Diante desse cenário, destaca-se a segurança da BIOS (*Basic Input/Output System*) sob a especificação UEFI (*Unified Extensible Firmware Interface*). Diferentemente da BIOS tradicional, a UEFI introduz melhorias modulares e recursos avançados

de segurança. Contudo, a proteção efetiva depende da integração de diversas camadas de defesa, incluindo senhas de BIOS, atualizações assinadas de fontes confiáveis, proteção da pilha que impede a execução de código, armazenamento criptografado, antivírus no sistema operacional e processos de desenvolvimento seguro desde a concepção do código-fonte, sendo esta última especialmente relevante para a proposta deste trabalho [Yao and Zimmer 2020].

Ainda assim, o *firmware* enfrenta ameaças sofisticadas e persistentes, difíceis de serem detectadas apenas com métodos tradicionais de teste [Richardson et al. 2019, UEFI 2021]. Essa complexidade exige métodos combinados de análises estáticas e dinâmicas, integrados a processos robustos como o *Development, Security, and Operations* (DevSecOps) e o *Secure Software Development Lifecycle* (SSDLC), para identificar vulnerabilidades desde os estágios iniciais do desenvolvimento.

As vulnerabilidades presentes na camada de *firmware* têm consequências particularmente graves, pois comprometem diretamente a base do sistema operacional [Machado 2018]. Assim, métodos proativos, como a instrumentação binária e técnicas avançadas de *fuzzing*, tornam-se indispensáveis para detectar falhas que só surgem durante a execução da BIOS. Contudo, apesar de sua eficácia comprovada em níveis superiores do sistema, essas técnicas ainda são raramente aplicadas em cenários pré-SO [Beekman 2015, Fioraldi et al. 2020].

Diante desse panorama, este estudo propõe um *pipeline* (sequência de etapas) automatizado que integra práticas DevSecOps especificamente voltadas à segurança dinâmica de BIOS UEFI. Utilizando tecnologias como Jenkins e Docker, o *pipeline* automatiza etapas críticas, incluindo compilação da BIOS, extração de módulos, instrumentação binária e testes dinâmicos via *fuzzing*, permitindo a detecção precoce de vulnerabilidades durante o ciclo de desenvolvimento.

A proposta é especialmente relevante devido à complexidade logística enfrentada por equipes distribuídas globalmente, que frequentemente precisam acessar dispositivos físicos específicos para realizar análises de segurança. Ao possibilitar testes em ambientes emulados, elimina-se a dependência física, ampliando a cobertura de segurança e acelerando a identificação de falhas. Além de reduzir custos operacionais, essa metodologia aprimora a robustez contra ameaças persistentes, contribuindo de forma econômica e prática para a segurança cibernética em um mercado cada vez mais interconectado e exigente.

Este trabalho norteia-se basicamente em três principais questões de pesquisa: (I) a efetividade do *pipeline* na detecção de vulnerabilidades em BIOS UEFI sem acesso a *hardware* físico; (II) a eficácia dos resultados obtidos em testes realizados em *firmware* de referência em comparação com aqueles realizados diretamente em BIOS reais; e (III) o custo-benefício da integração dessa abordagem a fluxos de integração contínua (*Continuous Integration* (CI)) distribuídos. Parte-se da hipótese de que um *pipeline* automatizado, integrando instrumentação binária e *fuzzing*, é capaz de identificar vulnerabilidades em módulos de BIOS, com sobrecarga aceitável e eficiência equivalente ou superior às abordagens tradicionais.

As principais contribuições deste estudo são: (I) desenvolvimento de um *pipeline* automatizado DevSecOps independente de *hardware*, integrando *fuzzing*, instrumentação

binária e emulação; (II) validação prática da abordagem em uma BIOS real com 645 módulos EFI, dos quais 34 (5,27%) apresentaram falhas operacionais; e (III) demonstração da capacidade da metodologia em reduzir significativamente a quantidade de módulos que exigem análise manual detalhada, economizando tempo e recursos humanos.

Este artigo está estruturado da seguinte forma: a Seção 2 descreve o Referencial Teórico, abordando conceitos de *firmware*, UEFI, DevSecOps, instrumentação binária e *fuzzing*. A Seção 3 apresenta a Revisão da Literatura, comparando as principais soluções existentes e situando a proposta desse trabalho. Na Seção 4 detalha-se a Metodologia, com o *pipeline* automatizado de testes dinâmicos em BIOS UEFI. A Seção 5 expõe os Resultados obtidos com a execução do *pipeline*. A Seção 6 dedica-se à Discussão dos achados e ao seu impacto prático. Por fim, a Seção 7 reúne as Considerações Finais, contemplando as Limitações do estudo, as recomendações para Trabalhos Futuros e a Conclusão.

2. Referencial Teórico

Nesta seção são apresentados os conceitos essenciais relacionados à segurança de *firmware*, destacando BIOS e UEFI, vulnerabilidades, práticas DevSecOps e SSDLC, técnicas de *fuzzing* e métodos de análise dinâmica com instrumentação binária.

2.1. Firmware, BIOS e UEFI

Firmware é o *software* de baixo nível responsável por inicializar o *hardware* antes que o sistema operacional assuma controle, operando sem as camadas defensivas típicas dos sistemas operacionais. Inicialmente, esse papel era desempenhado exclusivamente pelo BIOS (*Basic Input/Output System*), responsável por executar rotinas essenciais como a inicialização de memória e dispositivos periféricos [Machado 2018]. Com o avanço das plataformas computacionais, surgiu a necessidade de superar limitações do BIOS tradicional, como dificuldades na manutenção e portabilidade. Em resposta, foi desenvolvida a especificação UEFI (*Unified Extensible Firmware Interface*), que propõe uma arquitetura modular, segura e flexível. Esta especificação inclui funcionalidades como o *Secure Boot*, que protege contra ameaças de baixo nível ao verificar a integridade do *software* durante o processo de inicialização (*boot*), além das etapas padronizadas pela especificação *Platform Initialization* (PI), ilustradas na Figura 1 e detalhadas na Tabela 1 [UEFI 2021, UEFI PI 2024]. Uma implementação amplamente adotada da especificação UEFI é fornecida pelo projeto de código aberto *EFI Development Kit 2* (EDK2), cuja imagem de referência para máquinas virtuais é conhecida como *Open Virtual Machine Firmware* (OVMF).

2.2. Módulos EFI e Vulnerabilidades

Módulos *Extensible Firmware Interface* (EFI) (`.efi`) são componentes binários essenciais das fases PEI e DXE da UEFI/PI, responsáveis pela inicialização do *hardware* e serviços que antecedem o carregamento do sistema operacional. Devido à sua alta complexidade, esses módulos são frequentemente alvo de ataques explorando vulnerabilidades críticas, como a *BlackLotus*, que permite contornar o *Secure Boot*, e a recente *PixieFail*, que explora falhas em pilhas IPv6 [Binarly 2021, Quarkslab et al. 2024]. Essas vulnerabilidades destacam a necessidade de métodos eficazes, como *fuzzing* e análises dinâmicas, para detectar e mitigar tais riscos de segurança.

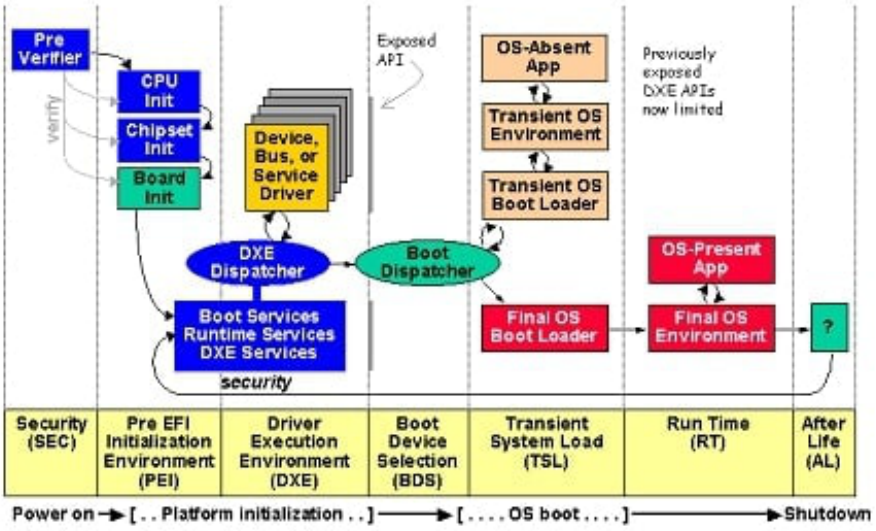

Figura 1. Fases de inicialização de um sistema PI/UEFI ([UEFI PI 2024]).

Tabela 1. Atribuições das fases do *boot* UEFI/PI.

Fase	Funções Principais
SEC	Inicialização básica do *hardware* e estabelecimento da base segura.
PEI	Ativação inicial da memória e localização de *firmwares* adicionais.
DXE	Carregamento de *drivers* e serviços essenciais para o sistema.
BDS	Seleção e inicialização do dispositivo que carregará o sistema operacional.

2.3. DevSecOps, SSDLC e *Pipeline* CI para *Firmware*

O desenvolvimento seguro de *firmware* enfrenta desafios devido à complexa cadeia de suprimentos, que começa com o desenvolvimento inicial em projetos como o EDK2 (por exemplo, via *Independent BIOS Vendors* (IBVs)), passando por *Original Design Manufacturers* (ODMs), até chegar aos *Original Equipment Manufacturers* (OEMs), responsáveis pelas personalizações finais antes de lançar os produtos ao mercado. Nesse cenário, princípios de DevSecOps, uma extensão do DevOps que incorpora práticas de segurança desde o início do ciclo de desenvolvimento, integrados à Integração Contínua (CI) e ao *Secure Software Development Lifecycle* (SSDLC), abordagem que busca garantir a segurança em todas as fases do ciclo de vida do *software*, tornam-se essenciais. *Pipelines* CI automatizados permitem executar testes de segurança durante todas as etapas do desenvolvimento, reduzindo a exposição de vulnerabilidades e facilitando correções antes do lançamento dos produtos finais. Esse processo é ilustrado na Figura 2.

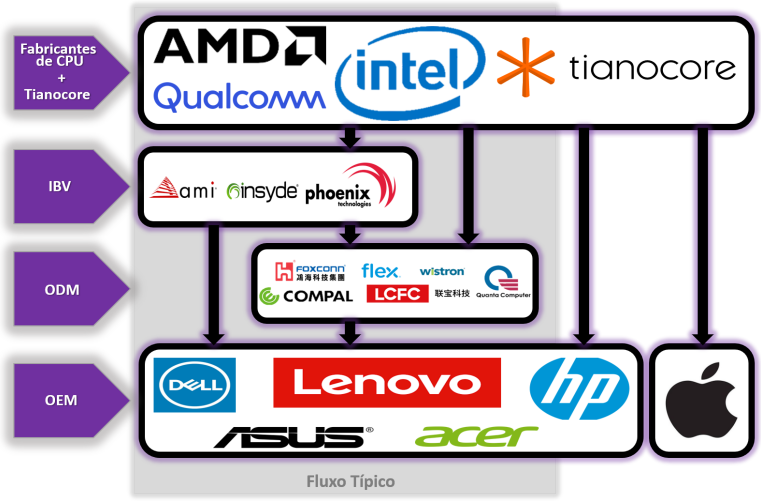

Figura 2. Cadeia de suprimentos de uma BIOS UEFI.

2.4. Instrumentação Binária, *Fuzzing*, Emulação e Análise Dinâmica em Módulos EFI

Embora análises estáticas sejam importantes, vulnerabilidades frequentemente só se manifestam durante a execução real. Neste contexto, a análise dinâmica torna-se essencial, permitindo identificar erros lógicos e falhas de tratamento de entradas específicas. Para realizar essas análises, técnicas como a instrumentação binária são utilizadas. Instrumentação binária consiste em inserir instruções adicionais em um código binário, possibilitando a monitoração detalhada da execução, facilitando a identificação de comportamentos anômalos ou vulnerabilidades.

Neste trabalho, emprega-se a ferramenta EFI-Fuzz, que utiliza *fuzzing*, uma técnica de testes que fornece entradas inválidas ou aleatórias para explorar comportamentos inesperados, aliada aos motores AFL++ e Unicorn. O Unicorn reutiliza o emulador de *Central Processing Unit* (CPU) do QEMU. A integração dessas ferramentas é complementada pelo *framework* Qiling [Carlsbad 2020]. A Figura 3 mostra o *roadmap* dessas ferramentas integradas.

É importante notar a distinção crítica entre emulação e simulação neste contexto. Enquanto simulação reproduz comportamentos externos de sistemas, a emulação (usada neste trabalho) recria fielmente o ambiente interno necessário para executar diretamente o código binário original dos módulos EFI. Tal abordagem fornece maior precisão na detecção de vulnerabilidades e comportamentos específicos que apenas simulações não capturariam.

Figura 3. Etapas para o UEFI Fuzzer.

Dessa forma, essa metodologia permite uma análise profunda, eficiente e independente do *hardware*, essencial para fortalecer a segurança do *firmware* em ambientes

complexos e distribuídos.

3. Revisão da Literatura

Embora a segurança em BIOS UEFI seja um tema relativamente recente, já existem contribuições significativas que abordam diversos aspectos relevantes dessa área. Richardson et al. [Richardson et al. 2019] apresentam o CHIPSEC, uma ferramenta que permite tanto a extração e análise de módulos EFI quanto testes diretamente no *host* em execução. O *Host-Based Firmware Analyzer* (HBFA), apresentado no mesmo trabalho, utiliza *fuzzing* e execução simbólica para detectar vulnerabilidades complexas durante o desenvolvimento da BIOS. Contudo, ambas as soluções exigem acesso direto ao *hardware* ou a ambientes específicos, limitando sua aplicabilidade em *pipelines* geograficamente distribuídos.

Yang et al. [Yang et al. 2020], por outro lado, utilizam a plataforma Simics para executar testes dinâmicos em módulos UEFI, incluindo técnicas de *fuzzing*, fornecendo uma visão detalhada dos módulos em execução. Esta abordagem complementa análises estáticas, mas não integra diretamente os resultados num fluxo automatizado de desenvolvimento contínuo.

Em linha semelhante, Yin et al. [Yin et al. 2023] propõem o RSFuzzer, técnica híbrida avançada de *fuzzing* voltada a vulnerabilidades profundas em manipuladores *System Management Interrupt* (SMI) (*SMI Handlers*). Apesar da sofisticação, sua aplicação é específica para certos módulos da BIOS.

Outro trabalho relevante é o de Häuser [Häuser 2023], que sugere um novo formato executável para módulos UEFI com foco em segurança estrutural, incluindo controle de fluxo de execução. Essa proposta fortalece a segurança preventiva, embora não contemple técnicas dinâmicas ou integração em DevSecOps.

Gomes et al. [Gomes et al. 2016] apresentam o UTTOS, que testa módulos UEFI usando recursos de sistemas operacionais convencionais. Essa abordagem facilita testes específicos, porém limita-se a validações no ambiente pós-SO.

Putra e Kabetta [Putra and Kabetta 2022], apesar de não abordarem especificamente *firmware*, demonstram o valor de integrar testes estáticos e dinâmicos em *pipelines* DevSecOps. Este trabalho traz *insights* relevantes sobre automação e integração contínua que são úteis ao cenário deste estudo.

Beekman [Beekman 2015], por sua vez, contribui de forma significativa com o desenvolvimento do EFIPERun, uma ferramenta pioneira em instrumentação e análise dinâmica pré-SO sem depender diretamente de *hardware*. Tal proposta é alinhada ao objetivo deste trabalho, apesar de não contemplar automação via DevSecOps.

Recentemente, Shafiuzzaman et al. [Shafiuzzaman et al. 2024] propuseram o STASE, combinando análise estática e execução simbólica para gerar assinaturas de vulnerabilidades automaticamente. Lu et al. [Lu et al. 2025] complementam com o EFIMemGuard, focado na detecção e mitigação automatizada de vulnerabilidades de segurança de memória por meio de análise estática, e também evidenciam a escassez reconhecida de bases públicas e rotuladas de BIOS. Já Matsuo et al. [Matsuo et al. 2024] sugerem o uso de chaves seladas via *Trusted Platform Module* (TPM) para ofuscação

de módulos *System Management Mode* (SMM), elevando a segurança contra ataques [Shafiuzzaman et al. 2024, Matsuo et al. 2024, Lu et al. 2025].

A proposta apresentada neste estudo complementa e amplia as abordagens existentes, combinando, de forma inédita, múltiplas técnicas de segurança: *fuzzing*, instrumentação binária e análise dinâmica, em uma integração completa num pipeline DevSecOps independente de *hardware*. Tal combinação proporciona vantagens concretas, especialmente a possibilidade de realizar avaliações contínuas sem depender do acesso físico aos dispositivos ou do código-fonte original. Essa abordagem reduz significativamente barreiras operacionais e logísticas enfrentadas por equipes distribuídas globalmente, representando um diferencial substancial frente aos trabalhos previamente publicados. A abordagem deste trabalho distingue-se pela integração de múltiplas técnicas de segurança, o que torna uma comparação quantitativa direta com estudos anteriores impraticável. Conforme sintetizado na Tabela 2, a estratégia proposta preenche lacunas não contempladas pelas abordagens existentes, e sua relevância é, portanto, demonstrada pela aplicabilidade contínua e pela eficiência operacional que oferece.

Tabela 2. Comparação de abordagens de segurança de *firmware*.

Trabalho	Estática	Dinâmica	Fuzzing	Pipeline CI	Containers	Instrumentação	HW-Indep.
EFIPERun [Beekman 2015]	–	✓	–	–	–	✓	✓
UTTOS [Gomes et al. 2016]	–	✓	–	–	–	–	✓
Chipsec e HBFA [Richardson et al. 2019]	✓	✓	✓	–	–	–	–
Simics [Yang et al. 2020]	–	✓	✓	–	–	✓	✓
DevSecOps [Putra and Kabetta 2022]	✓	✓	–	✓	✓	–	✓
RSFuzzer [Yin et al. 2023]	–	✓	✓	–	–	✓	✓
New File Format [Häuser 2023]	✓	–	–	–	–	–	✓
STASE [Shafiuzzaman et al. 2024]	✓	✓	–	–	–	✓	✓
SmmPack [Matsuo et al. 2024]	–	–	–	–	–	–	–
EFIMemGuard [Lu et al. 2025]	✓	–	–	–	–	–	✓
Este Trabalho	–	✓	✓	✓	✓	✓	✓

4. Metodologia

Nesta seção são descritos o método de pesquisa utilizado e o *pipeline* de testes dinâmicos proposto, detalhando suas principais etapas com o suporte de figuras para facilitar a compreensão do fluxo adotado.

4.1. Método de Pesquisa

O estudo realizado é exploratório e aplicado, com o objetivo de identificar lacunas relacionadas à segurança em *firmware* UEFI e propor soluções práticas por meio de automação e integração contínua.

4.2. Cenário Proposto

O cenário proposto envolve equipes de produto e *Original Design Manufacturers* (ODMs) que desenvolvem BIOS em máquinas locais, configuradas como agentes Jenkins. Esse cenário permite que a segurança do *firmware* seja testada automaticamente sem a necessidade de compartilhamento do código-fonte.

4.3. Ambiente Utilizado

Os testes foram conduzidos em um computador com processador Intel i7-10750H, 16 GB de RAM e sistema operacional Windows 11 Pro. Duas máquinas virtuais utilizando o VirtualBox foram configuradas: uma representando o servidor Jenkins (Linux Mint 21.2) e outra como agente Jenkins (Ubuntu 22.04). Ambas receberam 4 GB de RAM e 4 vCPUs, com Docker, Jenkins versão 2.141.3 e OpenJDK versão 17 instalados. O resumo das ferramentas integradas ao *pipeline* é exibido na Figura 4.

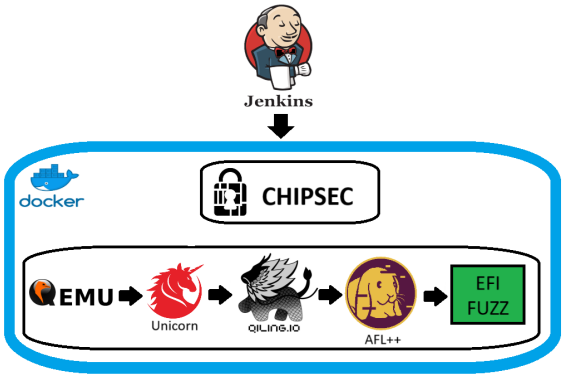

Figura 4. Etapas do *pipeline* de testes dinâmicos em BIOS UEFI.

4.4. *Pipeline* Jenkins

O *pipeline* implementado é composto por sete etapas, brevemente descritas a seguir:

a) **Checkout SCM:** realiza a sincronização automática do código-fonte utilizando o Git como ferramenta de SCM (Source Control Management), garantindo o uso da versão mais recente do projeto.
b) **Inicialização:** prepara o ambiente removendo *builds* anteriores e configurando permissões.
c) **Compilação da BIOS:** compila a BIOS utilizando o OVMF do projeto EDK2 para testes iniciais, gerando o arquivo binário necessário (conforme ilustrado na Figura 5).
d) **Construção do Contêiner Docker:** cria o contêiner Docker padronizado contendo as ferramentas Chipsec, EFI-Fuzz, AFL++ e dependências, proporcionando um ambiente uniforme para os testes (Figura 6).
e) **Extração dos Módulos EFI:** executa o Chipsec dentro do contêiner para decodificar o binário da BIOS e extrair módulos EFI (.efi), possibilitando uma análise individualizada (Figura 7).
f) **Instrumentação Binária e *Fuzzing*:** os módulos EFI são instrumentados utilizando EFI-Fuzz e submetidos a testes dinâmicos em ambiente emulado para identificação proativa de vulnerabilidades operacionais (Figura 8).
g) **Geração do Relatório Final:** consolida automaticamente os resultados obtidos nas etapas anteriores em um relatório sucinto, evidenciando os módulos aprovados e aqueles com falhas.

4.5. Adaptação para Testes Reais

Para validar a aplicabilidade prática, o *pipeline* foi adaptado para executar testes não apenas em BIOS compiladas via OVMF, mas também em imagem real de BIOS já compilada, ampliando assim o escopo e garantindo sua viabilidade técnica em cenários reais.

```
[Pipeline] { (#2 BIOS Compilation)
[Pipeline] echo
Step 2.1: Compiling BIOS...
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ ./Scripts/BuildOvmf.sh
Loading previous configuration from /home/user/edk2/Conf/BuildEnv.sh
Using EDK2 in-source Basetools
WORKSPACE: /home/user/edk2
EDK_TOOLS_PATH: /home/user/edk2/BaseTools
CONF_PATH: /home/user/edk2/Conf
Build environment: Linux-5.15.0-56-generic-x86_64-with-glibc2.35
Build start time: 01:53:19, Mar.17 2024
Processing meta-data .
Architecture(s)  = X64
Build target     = RELEASE
Toolchain        = GCC5

Active Platform          = /home/user/edk2/OvmfPkg/OvmfPkgX64.dsc
........ done!
Building ... /home/user/edk2/MdePkg/Library/UefiLib/UefiLib.inf [X64]
```

Figura 5. Compilação da BIOS via *pipeline*.

4.6. Técnicas Aplicadas

Além dos métodos técnicos descritos, foram aplicadas técnicas adicionais como revisão da literatura, análise comparativa de ferramentas existentes e testes em ambiente controlado visando aprimorar continuamente a eficiência do *pipeline* proposto.

5. Resultados

Esta seção apresenta os principais achados obtidos com a execução do *pipeline*, evidenciando sua eficácia na análise dinâmica de segurança em BIOS UEFI.

5.1. Execução do *Pipeline* e Análise Inicial

A execução completa do *pipeline*, representada na Figura 9, confirma a viabilidade da abordagem automatizada para testes de segurança. Os estágios principais, compilação do OVMF (Figura 5), construção do contêiner Docker (Figura 6), extração dos módulos EFI (Figura 7) e emulação dos módulos (Figura 8), ocorreram com sucesso e sem falhas de execução.

O tempo de execução variou conforme o desempenho da máquina hospedeira e a velocidade da conexão, especialmente na primeira execução, em que o carregamento de dependências é mais demorado. Nas execuções subsequentes, o uso de *cache* otimizou significativamente o desempenho.

5.2. Comparativo: OVMF vs. BIOS Real

Durante os testes com o *firmware* de referência, identificou-se que o OVMF, embora útil para provas de conceito, possui limitações no suporte às variáveis UEFI utilizadas pelo

```
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (#3 Docker container building)
[Pipeline] echo
Step 3.1: Building and running with the container via Dockerfile...
[Pipeline] script
[Pipeline] {
[Pipeline] isUnix
[Pipeline] withEnv
[Pipeline] {
[Pipeline] sh
+ docker build -t dynamictestscontainer -f ./Docker/Dockerfile ./Docker
DEPRECATED: The legacy builder is deprecated and will be removed in a future rele
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  8.704kB

Step 1/40 : FROM ubuntu:22.04
 ---> e4c58958181a
Step 2/40 : ENV DEBIAN_FRONTEND=noninteractive
 ---> Using cache
 ---> 8c72c0ba93fd
Step 3/40 : RUN apt-get update
 ---> Using cache
 ---> 4ce367148450
Step 4/40 : RUN apt-get install -y         build-essential         python3
pciutils         kmod         git         make         libssl-dev         bison
autoconf         libz3-dev         libboost-all-dev
 ---> Using cache
```

Figura 6. Construção do contêiner Docker via *pipeline*.

EFI-Fuzz. Como ilustrado na Figura 10, os arquivos essenciais para o *fuzzing* — *pickle* e entradas AFL — não são gerados adequadamente, o que impede a instrumentação completa.

O sucesso obtido com a BIOS real, em contrapartida, permitiu abordar diretamente a questão de pesquisa II deste estudo. Por meio dela, foi possível gerar todos os artefatos necessários para a emulação e execução dos testes (Figura 11), o que demonstra a aplicabilidade prática e a robustez do *pipeline* proposto em ambientes realistas.

5.3. Detecção de Anomalias em Módulo Modificado

Para demonstrar a capacidade de detecção de falhas, um módulo funcional (AbtSetup.efi) foi modificado manualmente com remoção de *bytes* em um editor hexadecimal. A execução do módulo original não apresentou erros, conforme Figura 12.

Já o módulo corrompido gerou falhas visíveis no terminal (Figura 13), indicadas por linhas iniciadas com "[x]". Essas falhas apontam comportamentos anômalos durante a execução, tais como desvios inesperados no fluxo original do módulo, condições inválidas que não deveriam ocorrer em operação normal, ou instruções ilegais decorrentes da corrupção binária introduzida manualmente. Isso demonstra que a metodologia adotada consegue detectar de forma automática interrupções ou inconsistências na execução dos módulos EFI, destacando potenciais pontos de vulnerabilidade operacional.

```
[Pipeline] { (#4 Extraction of EFI modules)
[Pipeline] echo
Step 4.1: Running the chipsec extractor and filtering .efi modules...
[Pipeline] script
[Pipeline] sh
+ sudo docker run --rm -v /home/user/edk2/Build/OvmfX64/RELEASE_GCC5/FV:/
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version  : 1.12.9
[CHIPSEC] Arguments: uefi decode OVMF.fd
```

Figura 7. Extração dos módulos EFI via *pipeline*.

```
[Pipeline] { (#5 Binary instrumentation and dynamic testing)
[Pipeline] echo
Step 5.1: Preparing for dynamic testing...
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ sudo docker run --rm -v /home/user/edk2/Build/OvmfX64/RELEASE_GCC5/F
Creating_nvram_pickle
[*] Pickled variable space
[!] Done!
Creating_afl_inputs
Writing space.
Instrumenting_efi
[95m[+][0m      Profile: default
[94m[=][0m      DXE heap at 0x04000000
[94m[=][0m      DXE stack at 0x0507fff0
```

Figura 8. Execução de módulo EFI instrumentado via *pipeline*.

5.4. Execução Abrangente em BIOS Real

Aplicando o *pipeline* à BIOS real, todos os 645 módulos EFI presentes no *firmware* foram executados com sucesso, sendo possível realizar os testes dinâmicos completos em cada um deles. A Tabela 3 resume os dados obtidos.

Dos módulos executados, 34 apresentaram falhas, o que representa 5,27% do total. Isso responde positivamente à questão de pesquisa I, sobre a efetividade do *pipeline*. Conforme ilustrado na Figura 14, esse dado reforça a importância da execução dinâmica como forma de identificar potenciais vulnerabilidades que poderiam passar despercebidas em testes estáticos.

6. Discussão

Os resultados obtidos demonstram que a adoção de um *pipeline* automatizado, executado em contêineres Docker, oferece uma abordagem eficaz, flexível e independente de *hardware*, promovendo integração contínua em ambientes diversos. Tal independência

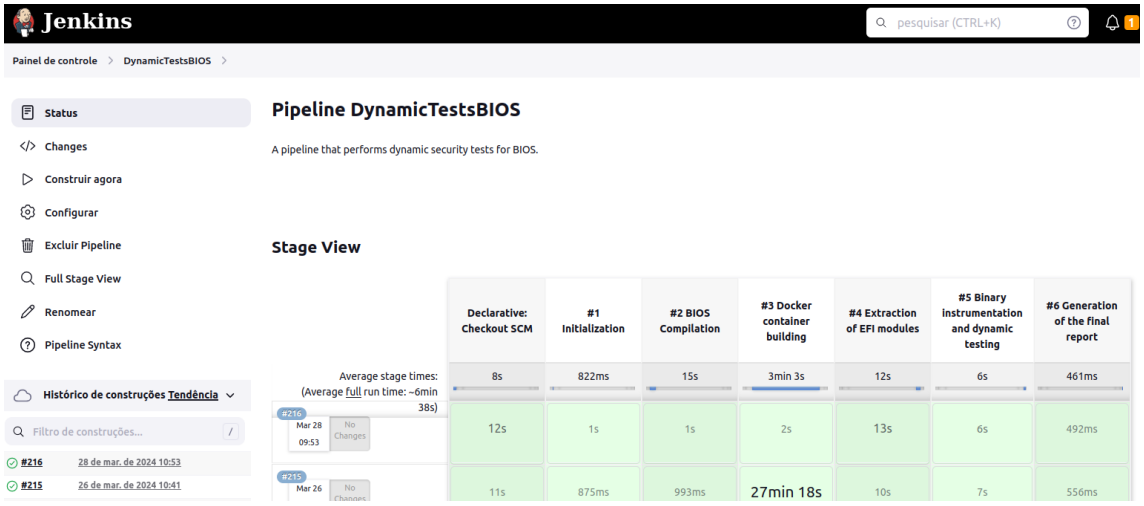

Figura 9. Tela de visualização geral das execuções do *pipeline*.

```
[Pipeline] { (#5 Binary instrumentation and dynamic testing)
[Pipeline] echo
Step 5.1: Preparing for dynamic testing...
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ sudo docker run --rm -v /home/user/edk2/Build/OvmfX64/RELEASE_GCC5/FV:
Creating_nvram_pickle
[*] Pickled variable space
[!] Done!
The complete pickled NVRAM environment can be found here: nvram.pickle
Creating_afl_inputs
Writing space.
```

Figura 10. Pickle e AFL Inputs para o binário do OVMF.

```
Step 5.1: Preparing for dynamic testing...
[Pipeline] script
[Pipeline] {
[Pipeline] sh
+ sudo docker run --rm -v /home/user/edk2/Build
Creating nvram pickle
[*] Pickled variable StdDefaults
[*] Pickled variable DeploymentModeNv
[!] Done!
Creating_afl_inputs
[*] Wrote StdDefaults/StdDefaults_0
[*] Wrote DeploymentModeNv/DeploymentModeNv_0
[!] Done!
```

Figura 11. Pickle e AFL Inputs para o binário de uma BIOS real.

operacional reduz barreiras logísticas e técnicas, permitindo que equipes distribuídas globalmente realizem testes de segurança sem acesso físico ao dispositivo.

A principal contribuição reside na capacidade de executar módulos de BIOS reais

```
[94m[=][0m      Enabling sanitizers ['memory']
[95m[+][0m      0x00000000040001b8: LocateProtocol(Protocol = EfiHiiStringProtocolGuid, Regist
[95m[+][0m      0x00000000040001b8: LocateProtocol(Protocol = EfiHiiDatabaseProtocolGuid, Regi
[95m[+][0m      0x00000000040001b8: LocateProtocol(Protocol = EfiHiiConfigRoutingProtocolGuid,
[95m[+][0m      0x00000000040001b8: LocateProtocol(Protocol = EfiHiiFontProtocolGuid, Registra
[95m[+][0m      0x00000000040001b8: LocateProtocol(Protocol = EfiHiiImageProtocolGuid, Registr
[95m[+][0m      0x00000000040001b8: LocateProtocol(Protocol = b0688a59-6a13-4f89-8ca4-84f4bf36
[95m[+][0m      0x0000000004000238: GetVariable(VariableName = L"AbtSetup", VendorGuid = c7f83
0x0507ffa8, Data = 0x00104700) = 0x0
[94m[=][0m      Installing protocol interface fa02fb02-c112-4d4e-b29b-347a446ae9ad to 0x1046f0
[95m[+][0m      0x00000000040001c0: InstallMultipleProtocolInterfaces(Handle = 0x00104670) = 0
[95m[+][0m      0x0000000004000190: OpenProtocol(Handle = 0x00100000, Protocol = EfiHiiPackage
ControllerHandle = NULL, Attributes = 0x2) = 0x8000000000000000e
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (#6 Generation of the final report)
```

Figura 12. Execução de um módulo EFI normal.

Tabela 3. Sumário da Execução dos Módulos EFI de uma BIOS Real.

Descrição	**Quantidade/Tempo**
Número total de módulos EFI testados	645
Número total de variáveis EFI serializadas	619
Tempo total de execução	47min 45s

em ambiente emulado, viabilizando testes dinâmicos precoces e aumentando a profundidade da análise, mesmo sem o código-fonte. Essa abordagem complementa as análises estáticas tradicionalmente utilizadas, ampliando a cobertura de segurança ao identificar falhas que só se manifestam em tempo de execução. Embora uma tentativa inicial com *firmware* OVMF tenha evidenciado limitações devido à ausência de variáveis EFI necessárias, prejudicando a instrumentação e o *fuzzing*, a execução subsequente com uma BIOS real confirmou plenamente a eficácia do método. A inclusão desta análise serve para documentar os desafios práticos envolvidos, contribuindo para a reprodutibilidade de experimentos na área ao informar sobre as limitações do OVMF para este tipo de aplicação.

Quanto à escalabilidade, os resultados foram bastante positivos. Um tempo de execução de 47 minutos e 45 segundos para analisar 645 módulos de uma BIOS real demonstra uma alta escalabilidade, projetando uma execução viável em menos de 8 horas mesmo para *firmwares* dez vezes maiores. Essa eficiência, relevante para a questão de pesquisa III, permite gerar rapidamente um escopo refinado de módulos prioritários para análise manual e otimizar o esforço da equipe técnica especializada. A economia de tempo e a redução do esforço em tarefas repetitivas representam benefícios práticos claros, tornando a adoção desse *pipeline* altamente compensatória em cenários reais.

Além disso, o *pipeline* proposto apresenta forte potencial de impacto prático: possibilita redução de custos com infraestrutura e retrabalho, ao mesmo tempo que fortalece a postura de segurança da empresa diante de *stakeholders* e usuários finais. Ao utilizar ferramentas *open-source*, amplia-se também a viabilidade econômica da solução.

```
[91m[x][0m      Memory map:
[91m[x][0m      Start        End          Perm    Label         Image
[91m[x][0m      0000100000 - 000010e000   rwx     [module]      AbtSetup_Crashed.efi
[91m[x][0m      0004000000 - 0004001000   rwx     [heap]
[91m[x][0m      0004001000 - 0004011000   rwx     [heap]
[91m[x][0m      0004011000 - 0004012000   rwx     [heap]
[91m[x][0m      0004012000 - 0004014000   rwx     [heap]
[91m[x][0m      0005000000 - 0005080000   rwx     [stack]
[91m[x][0m      0070008000 - 0070010000   ---     [SMRAM SSA]
[91m[x][0m      0077000000 - 0077001000   rwx     [heap]
[91m[x][0m      0077001000 - 0077011000   rwx     [heap]
[91m[x][0m      0077011000 - 0077012000   rwx     [heap]
[91m[x][0m      0077ff0000 - 0078000000   rwx     [stack]
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (#6 Generation of the final report)
```

Figura 13. Execução de um módulo EFI corrompido.

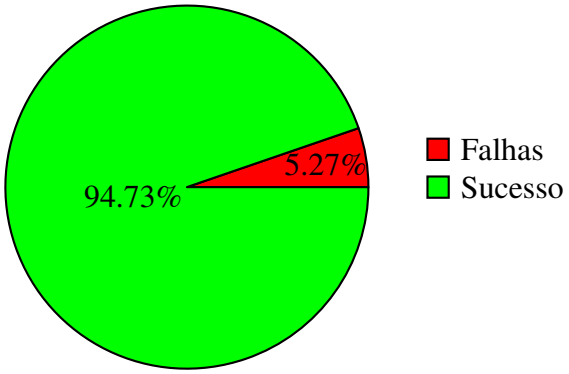

Figura 14. Distribuição dos Resultados dos Testes de Módulos EFI.

Ainda que automatizada, a proposta não elimina a importância da análise especializada. O julgamento humano continua essencial para interpretar falhas, evitar falsos positivos e orientar a mitigação adequada dos problemas encontrados.

7. Considerações Finais

7.1. Limitações

Dentre as limitações, destaca-se a cobertura parcial da fuzzificação, principalmente devido à dependência entre módulos EFI, o que dificulta a execução isolada de certos componentes. Outra restrição relevante refere-se à complexidade técnica do relatório final gerado pelo *pipeline* proposto, atualmente compreensível apenas por profissionais com elevado conhecimento em cibersegurança de *firmware*, limitando a disseminação e interpretação dos resultados. Além disso, embora a emulação seja fundamental para o método apresentado, ela pode não replicar perfeitamente todas as interações específicas

de *hardware* que ocorrem em ambientes físicos reais, potencialmente deixando alguns cenários de execução descobertos.

Outro aspecto importante é que o contêiner utilizado carece de medidas avançadas de *hardening*, o que representa um risco significativo, sobretudo em ambientes sensíveis. Exemplos críticos de tais medidas incluem restrições severas de rede e controles avançados de acesso às informações sensíveis dos testes realizados. Um ambiente de testes comprometido poderia resultar na exposição indevida dos binários analisados ou em interferências maliciosas nos resultados, comprometendo a confiabilidade dos testes realizados. Tal cenário demanda esforços adicionais de mitigação, como auditorias periódicas, controle rigoroso de acessos e uso de técnicas robustas de isolamento e criptografia, que atualmente não estão plenamente implementadas.

Finalmente, destaca-se uma limitação inerente ao escopo deste estudo: embora o *pipeline* identifique os módulos com falhas, a análise detalhada de seus tipos específicos ainda demanda um esforço manual significativo. Soma-se a isso um desafio sistêmico da área, que é a ausência de bases de dados públicas e rotuladas de BIOS, o que dificulta a realização de avaliações comparativas robustas e reprodutíveis.

7.2. Futuros Trabalhos

Como continuidade, recomenda-se inicialmente uma investigação mais detalhada e automatizada sobre as falhas específicas identificadas durante a execução dos testes. Tal estudo permitiria a criação de padrões e categorias de falhas, potencializando a eficácia das correções implementadas pelos desenvolvedores. Adicionalmente, sugere-se explorar futuramente técnicas avançadas de inteligência artificial que possam auxiliar na seleção dos módulos mais suscetíveis a falhas, aumentando assim a eficiência e assertividade na identificação proativa de vulnerabilidades.

Adicionalmente, propõe-se estimular iniciativas comunitárias voltadas à criação de conjuntos públicos de *firmwares* rotulados com vulnerabilidades comprovadas ou corrigidas. Isso possibilitaria *benchmarks* mais robustos e avaliações comparativas mais efetivas entre abordagens propostas na literatura científica.

Também é imperativo o aperfeiçoamento das técnicas de instrumentação e fuzzificação, buscando viabilizar testes que considerem as dependências entre módulos EFI, aumentando significativamente a abrangência e profundidade das análises. Quanto à segurança operacional do *pipeline*, recomenda-se implementar práticas rigorosas de *hardening* no contêiner utilizado, especialmente controles avançados de acesso, restrições rigorosas de rede, remoção de pacotes desnecessários e emprego de criptografia robusta, garantindo assim a integridade e confidencialidade dos dados analisados.

Por último, sugere-se o desenvolvimento de relatórios simplificados e acessíveis, acompanhados por *dashboards* interativos e intuitivos, capazes de atender não apenas especialistas em segurança, mas também desenvolvedores, equipes técnicas e gerenciais. Esses *dashboards* facilitariam a visualização rápida dos resultados, contribuindo para tomadas de decisão mais eficazes e promovendo uma cultura de segurança colaborativa e inclusiva dentro das organizações.

7.3. Conclusão

Este trabalho apresentou uma proposta de *pipeline* DevSecOps voltado à análise dinâmica de BIOS sob a especificação UEFI. Ao integrar compilação, extração, instrumentação e testes com ferramentas abertas em um fluxo automatizado, a metodologia demonstrou ser viável, segura e adaptável. O uso de emulação e contêineres ampliou a acessibilidade da análise, promovendo segurança desde os estágios iniciais do desenvolvimento, sem depender de acesso ao *hardware*.

Assim, este trabalho cumpre com os objetivos inicialmente propostos, destacando-se pelas seguintes contribuições: criação de um *pipeline* automatizado independente de *hardware*; validação prática da metodologia em BIOS real, identificando falhas operacionais em mais de 5% dos módulos testados; e a redução do esforço humano por meio da priorização de análises manuais posteriores.

Com isso, contribui-se não apenas para a segurança técnica dos dispositivos, mas também para a redução de custos, aumento da eficiência operacional e incentivo à adoção de boas práticas em toda a cadeia de desenvolvimento de *firmware*. A proposta se posiciona como um passo relevante na evolução dos testes de segurança de BIOS, com potencial de aplicação ampla em ambientes reais.

8. Agradecimentos

Os autores agradecem à Lenovo pelo suporte institucional durante o desenvolvimento do trabalho.

Referências

Beekman, J. (2015). Reverse engineering UEFI firmware. `https://jbeekman.nl/blog/2015/03/reverse-engineering-uefi-firmware/`. Acessado em 12/05/2025.

Binarly (2021). The firmware supply-chain security is broken: Can we fix it? *Binarly REsearch Team*. `https://www.binarly.io/blog/the-firmware-supply-chain-security-is-broken-can-we-fix-it`. Acessado em 12/05/2025.

Carlsbad, A. (2020). Moving from dynamic emulation of UEFI modules to coverage-guided fuzzing of UEFI firmware. `https://www.sentinelone.com/labs/moving-from-dynamic-emulation-of-uefi-modules-to-coverage-guided-fuzzing-of-uefi-firmware/`. Acessado em 12/05/2025.

Fioraldi, A., Maier, D., Eißfeldt, H., and Heuse, M. (2020). AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies*, Virtual Event. USENIX Association.

Gomes, E., Amora, P., Teixeira, E. M., Lima, A., Brito, F. T., Ciocari, J., and Machado, J. C. (2016). UTTOS: A tool for testing UEFI code in OS environment. In *28th IFIP International Conference on Testing Software and Systems*, volume 9976, pages 218–224, Graz, Austria. Springer.

Häuser, M. (2023). Designing a secure and space-efficient executable file format for the unified extensible firmware interface. Master's thesis, University of Kaiserslautern-Landau, Kaiserslautern, Germany.

Lu, Z., an Tan, Y., Cheng, X., Zheng, Z., Shi, N., and Li, Y. (2025). An automated framework for detecting and mitigating memory safety vulnerabilities in UEFI firmware. *Computers and Electrical Engineering*, 122:109945.

Machado, R. R. (2018). Desenvolvimento das fundações para acessibilidade em ambiente pré-OS. Master's thesis, Universidade Federal de São Carlos - UFSCar, Sorocaba, SP.

Matsuo, K., Tanda, S., Suzaki, K., Kawakoya, Y., and Mori, T. (2024). SmmPack: Obfuscation for SMM modules with TPM sealed key. In *21st International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, page 439–459, Lausanne, Switzerland. Springer.

Putra, A. M. and Kabetta, H. (2022). Implementation of DevSecOps by integrating static and dynamic security testing in CI/CD pipelines. In *International Conference of Computer Science and Information Technology*, pages 1–6, Laguboti, North Sumatra, Indonesia. IEEE.

Quarkslab, Falcon, F., and Arce, I. (2024). PixieFail: Nine vulnerabilities in Tianocore's EDK II IPv6 network stack. `https://blog.quarkslab.com/pixiefail-nine-vulnerabilities-in-tianocores-edk-ii-ipv6-network-stack.html`. Acessado em 12/05/2025.

Richardson, B., Wu, C., Yao, J., and Zimmer, V. J. (2019). Using host-based firmware analysis to improve platform resiliency. Technical report, Intel, S.l. `https://www.intel.com/content/dam/develop/external/us/en/documents/intel-usinghbfatoimproveplatformresiliency.pdf`. Acessado em 12/05/2025.

Shafiuzzaman, M., Desai, A., Sarker, L., and Bultan, T. (2024). STASE: Static analysis guided symbolic execution for UEFI vulnerability signature generation. In *39th International Conference on Automated Software Engineering*, pages 1783–1794, Sacramento, CA, USA. ACM.

UEFI (2021). UEFI forum - unified extensible firmware interface specification. `https://uefi.org/`. Acessado em 12/05/2025.

UEFI PI (2024). UEFI platform initialization specification. `https://uefi.org/sites/default/files/resources/PI_Spec_1_8_A_final_2024.03.05.pdf`. Version 1.8 Errata A; Acessado em 12/05/2025.

Yang, Z., Viktorov, Y., Yang, J., Yao, J., and Zimmer, V. (2020). UEFI firmware fuzzing with Simics virtual platform. In *57th Design Automation Conference*, pages 1–6, San Francisco, CA, USA. IEEE.

Yao, J. and Zimmer, V. (2020). *Building Secure Firmware: Armoring the Foundation of the Platform*. Apress, Berkeley, CA.

Yin, J., Li, M., Li, Y., Yu, Y., Lin, B., Zou, Y., Liu, Y., Huo, W., and Xue, J. (2023). RSFuzzer: Discovering deep SMI handler vulnerabilities in UEFI firmware with hybrid fuzzing. In *44th Symposium on Security and Privacy*, pages 2155–2169, San Francisco, CA, USA. IEEE.