



**UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA**

Henrique Borges Campelo

Detecção e Identificação de Comportamento Evasivo em
Executáveis para Windows Usando Instrumentação Binária
Dinâmica

**Manaus
2026**

Henrique Borges Campelo

Detecção e Identificação de Comportamento Evasivo em
Executáveis para Windows Usando Instrumentação Binária
Dinâmica

Dissertação apresentada ao Programa de
Pós-Graduação em Informática do Insti-
tuto de Computação da Universidade Fe-
deral do Amazonas como requisito para ob-
tenção do grau de Mestre em Informática.

Orientador: Prof. Eduardo Luzeiro Feitosa

Manaus
2026

Ficha Catalográfica

Elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

C193d Campelo, Henrique Borges
 Detecção e identificação de comportamento evasivo em executáveis para
 Windows usando instrumentação binária dinâmica / Henrique Borges
 Campelo. - 2026.
 113 f. : il., color. ; 31 cm.

 Orientador(a): Eduardo Luzeiro Feitosa.
 Dissertação (mestrado) - Universidade Federal do Amazonas, Programa
 de Pós-Graduação em Informática, Manaus, 2026.

 1. Malware evasivo. 2. Instrumentação Binária Dinâmica (DBI). 3.
 Detecção de técnicas. 4. Ferramenta de análise. 5. Engenharia reversa. I.
 Feitosa, Eduardo Luzeiro. II. Universidade Federal do Amazonas.
 Programa de Pós-Graduação em Informática. III. Título



Ministério da Educação
Universidade Federal do Amazonas
Coordenação do Programa de Pós-Graduação em Informática

FOLHA DE APROVAÇÃO

"DETECÇÃO E IDENTIFICAÇÃO DE COMPORTAMENTO EVASIVO EM EXECUTÁVEIS PARA WINDOWS USANDO INSTRUMENTAÇÃO BINÁRIA DINÂMICA"

HENRIQUE BORGES CAMPELO

Dissertação de Mestrado defendida e aprovada pela banca examinadora constituída pelos Professores:

Prof. Dr. Eduardo Luzeiro Feitosa - **Presidente**

Prof. Dr. Eduardo James Pereira Souto - **Membro Interno**

Prof. Dr. Gilbert Breves Martins - **Membro Externo**

Manaus, 20 de Março de 2026.



Documento assinado eletronicamente por **Eduardo James Pereira Souto, Professor do Magistério Superior**, em 10/04/2026, às 10:58, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Gilbert Breves Martins, Usuário Externo**, em 16/04/2026, às 10:27, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



Documento assinado eletronicamente por **Eduardo Luzeiro Feitosa, Professor do Magistério Superior**, em 16/04/2026, às 13:36, conforme horário oficial de Manaus, com fundamento no art. 6º, § 1º, do [Decreto nº 8.539, de 8 de outubro de 2015](#).



A autenticidade deste documento pode ser conferida no site https://sei.ufam.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3104161** e o código CRC **9D735DF3**.

Avenida General Rodrigo Octávio, 6200 - Bairro Coroado I Campus Universitário Senador Arthur Virgílio Filho, Setor Norte - Telefone: (92) 3305-1181 / Ramal 1193
CEP 69080-900, Manaus/AM, coordenadorppgi@icomp.ufam.edu.br

Referência: Processo nº 23105.013890/2026-94

SEI nº 3104161

Agradecimentos

Agradeço aos meus pais, Soares Borges da Silva e Maria Socorro Santiago Campelo, por sempre apoiarem minha educação e por me ensinarem que o conhecimento abre muitas portas na vida.

Agradeço à minha esposa, Kenya Alejandra Perez Rivas, e à minha filha, Katheryn Vanessa Perez Campelo, pelo amor, apoio, paciência e compreensão demonstrados durante todo o período de desenvolvimento desta pesquisa. Agradeço também ao meu irmão, Savio Borges Campelo, pelos conselhos, incentivo e motivação.

Aos meus colegas de estudo da Universidade Federal do Amazonas, agradeço pelas horas compartilhadas em atividades acadêmicas, estudos e trabalhos. Estendo meus agradecimentos a todos os professores que contribuíram para minha formação, em especial ao meu orientador Eduardo Luzeiro Feitosa, pelo constante suporte, orientação e confiança ao longo desta pesquisa.

Por fim, agradeço à Universidade Federal do Amazonas, aos seus colaboradores, técnicos e à seção administrativa, pelo apoio institucional e pelas condições oferecidas para a realização deste trabalho.

Resumo

No cenário tecnológico atual, marcado pela evolução contínua e pelo aprimoramento das ameaças digitais, a análise de software malicioso torna-se fundamental para o desenvolvimento de novas estratégias de defesa, capazes de proteger eficientemente os sistemas informáticos. Para esse propósito, a análise por meio de engenharia reversa é uma técnica crucial. É imperativo compreender as estratégias evasivas adotadas por *malwares* para otimizar o processo de análise, pois estas são projetadas para evadir os métodos convencionais de análise, exigindo abordagens mais sofisticadas para identificação e neutralização. Esta pesquisa foca na compreensão, detecção e mitigação das principais técnicas evasivas usadas por *malwares* em sistemas Windows, bem como na identificação das ferramentas de proteção empregadas por criadores de *malwares*, como *Themida* e *Obsidium*, que visam obstruir e complicar a análise por especialistas e ferramentas de segurança, permitindo uma delimitação mais precisa do escopo de análise e garantindo uma resposta mais eficaz contra comportamentos maliciosos. O objetivo principal deste trabalho é desenvolver estratégias para detectar táticas evasivas utilizadas por softwares maliciosos e identificar as ferramentas de proteção utilizadas, com o intuito de aprimorar e agilizar os processos de detecção e análise de *malwares*. Para alcançar tal objetivo, este trabalho também apresenta a **ContradeF**, uma ferramenta de instrumentação binária dinâmica construída sobre o *Intel Pin*, projetada especificamente para **observar**, **registrar** e **mitigar** o comportamento evasivo de binários Windows durante a execução. A ferramenta registra indícios de comportamento evasivo além de preservar a transparência do ambiente. A avaliação foi conduzida com amostras reais protegidas e executadas em ambiente virtualizado semelhante ao utilizado por analistas, demonstrando que a *ContradeF* é capaz de revelar o fluxo de execução oculto, reduzir o impacto de sondagens de ambiente e produzir trilhas de análise em nível de API, instrução e memória. As principais contribuições são: (i) a sistematização de técnicas evasivas contemporâneas; (ii) uma abordagem prática para apoiar pesquisadores e profissionais de segurança na análise de *malware* evasivo; e (iii) uma análise de algumas das estratégias de anti-análise dos protetores *Themida* e *Obsidium*.

Palavras-chave: *Malware* evasivo, Instrumentação Binária Dinâmica (*DBI*), detecção de técnicas, ferramenta de análise, engenharia reversa.

Abstract

In the current technological landscape, characterized by the continuous evolution and sophistication of digital threats, malware analysis has become fundamental for developing new defense strategies capable of efficiently protecting information systems. For this purpose, analysis via reverse engineering is a crucial technique. It is imperative to understand the evasive strategies adopted by *malware* to optimize the analysis process, as these are designed to evade conventional analysis methods, demanding more sophisticated approaches for identification and neutralization. This research focuses on the understanding, detection, and mitigation of the primary evasive techniques used by *malware* on Windows systems, as well as on identifying the protection tools employed by *malware* creators, such as *Themida* and *Obsidium*. These tools aim to obstruct and complicate analysis by experts and security tools, allowing for a more precise delimitation of the analysis scope and ensuring a more effective response against malicious behaviors. The main objective of this work is to develop strategies to detect evasive tactics used by malicious software and to identify the protection tools used, in order to enhance and streamline *malware* detection and analysis processes. To achieve this objective, this work also presents **Contradef**, a dynamic binary instrumentation (DBI) tool built on *Intel Pin*, specifically designed to **observe**, **log**, and **mitigate** the evasive behavior of Windows binaries during execution. The tool logs indicators of evasive behavior while preserving the transparency of the environment. The evaluation was conducted with real, protected samples executed in a virtualized environment similar to that used by analysts, demonstrating that *Contradef* is capable of revealing the hidden execution flow, reducing the impact of environment probing, and producing analysis traces at the API, instruction, and memory levels. The main contributions are: (i) the systematization of contemporary evasive techniques; (ii) a practical approach to support researchers and security professionals in the analysis of evasive *malware*; and (iii) an analysis of some of the anti-analysis strategies of the *Themida* and *Obsidium* protectors.

Keywords: Evasive *malware*, Dynamic Binary Instrumentation (DBI), technique detection, analysis tool, reverse engineering.

Lista de Figuras

2.1	(a) Código <i>assembly</i> x86 original, (b) Código reordenado usando instrução de salto incondicional. Singh and Singh (2018)	11
2.2	Inserção de código redundante (código lixo). Singh and Singh (2018)	12
4.1	Exemplo do uso de instruções de salto com o mesmo alvo. (Sikorski and Honig, 2012)	29
4.2	Instrução <i>jz</i> seguida por uma instrução <i>jnz</i> . (Sikorski and Honig, 2012)	29
4.3	Exemplo do uso de instruções com condição constante. (Sikorski and Honig, 2012)	30
4.4	Falso condicional de <i>xor</i> seguido por uma instrução <i>jz</i> . (Sikorski and Honig, 2012)	30
4.5	Exemplo de resolução padrão da <i>API</i> . (Cheng et al., 2021)	31
4.6	Chaves de registro usadas pelo <i>Themida</i> para detectar o <i>VMware</i> e o <i>VirtualBox</i> (Lee et al., 2021).	38
4.7	Técnica de detecção através da <i>API RegOpenKeyExA</i> (Lee et al., 2021).	38
4.8	Técnica de detecção do <i>VMware</i> utilizando informações da tabela de <i>firmware</i> (Lee et al., 2021).	39
4.9	Técnica de detecção do <i>VirtualBox</i> utilizando informações da tabela de <i>firmware</i> (Lee et al., 2021).	39
4.10	Técnica de detecção através da <i>API WideCharToMultiByte</i> (VBoxService.exe) (Lee et al., 2021).	40
4.11	Detecção através do <i>driver</i> VBoxGuest (Lee et al., 2021).	41
4.12	Detecção de Processo Pai (Polino et al., 2017b)	42
4.13	Detecção através da <i>API ZwQueryInformationProcess</i> (Lee et al., 2021).	44
4.14	Mitigando a técnica de detecção por uma operação XOR com trap flag (Lee et al., 2021).	46
5.1	Arquitetura da Ferramenta <i>Contradef</i>	52
5.2	Padrão de sequência de instruções do Obsidium v1.7.3 build 5 decompilada pelo <i>Ghidra</i>	54
5.3	Padrão de sequência de instruções do Obsidium v1.8.5 build 4 decompilada pelo <i>Ghidra</i>	54
5.4	Padrão de sequência de instruções do Themida, comparando diferentes versões no <i>Ghidra</i>	55
5.5	Captura de informações da função <i>NtQueryInformationProcess</i> .	57
5.6	Trecho de log produzido pelo módulo <i>TraceFcnCall</i> (M1).	58
5.7	Exemplo de log produzido pelo módulo <i>TraceFcnCall</i> M2	59
5.8	Extrato capturado pelo <i>TraceMemory</i> contendo dados da memória.	59

5.9	Fila de Instruções Instrumentadas e Sequências de Instruções a Serem Correspondidas	62
7.1	Teste de chamadas recorrentes à função <code>GetTickCount</code>	80
7.2	Teste de execução da função <code>GetTickCount</code> invocada 6 vezes sendo instrumentada pelo o <code>Pin</code>	81
7.3	Teste de execução de 1000 iterações da função <code>GetTickCount</code> invocada 6 vezes sendo instrumentada pelo o <code>Pin</code>	81
7.4	Execução da técnica de detecção de depuradores usando a função <code>RaiseException</code>	83

Nomenclatura

ACPI Interface Avançada de Configuração e Energia (*Advanced Configuration and Power Interface*)

API Interface de Programação de Aplicações (*Application Programming Interface*)

ASCII Código Padrão Americano para o Intercâmbio de Informação (*American Standard Code for Information Interchange*)

BIOS Sistema Básico de Entrada/Saída (*Basic Input/Output System*)

CE Critérios de Exclusão (*Exclusion Criteria*)

CI Critérios de Inclusão (*Inclusion Criteria*)

CPU Unidade Central de Processamento (*Central Processing Unit*)

DBI Instrumentação Binária Dinâmica (*Dynamic Binary Instrumentation*)

DIE Ferramenta para Identificação de Tipos de Arquivo Executáveis (*Detect It Easy*)

DLL Biblioteca de Vínculo Dinâmico (*Dynamic-Link Library*)

DMTF Força-Tarefa de Gerenciamento Distribuído (*Distributed Management Task Force*)

DSI Detector de Sequências de Instruções (*Instruction Sequence Detector*)

ERP Planejamento de Recursos Empresariais (*Enterprise Resource Planning*)

GQM Meta–Questão–Métrica (*Goal–Question–Metric*)

GUI Interface Gráfica do Usuário (*Graphical User Interface*)

HWND *Handle* de Janela (*Window Handle*)

IAT Tabela de Endereços de Importação (*Import Address Table*)

INT Interrupção (*Interrupt*)

JIT *Just-in-Time*

JSON Notação de Objetos JavaScript (*JavaScript Object Notation*)

MAC Endereço de Controle de Acesso ao Meio (*Media Access Control*)

NOP Instrução de Não Operação (*No Operation*)

<i>OEP</i>	Ponto de Entrada Original (<i>Original Entry Point</i>)
<i>OPCODE</i>	Código de Operação (<i>Operation Code</i>)
<i>PE</i>	Executável Portável (<i>Portable Executable</i>)
<i>PEB</i>	Bloco de Ambiente do Processo (<i>Process Environment Block</i>)
<i>PID</i>	Identificador de Processo (<i>Process ID</i>)
<i>PIN</i>	Framework de Instrumentação Binária Dinâmica da Intel (<i>PIN</i>)
<i>PoC</i>	Prova de Conceito (<i>Proof of Concept</i>)
<i>QBDI</i>	Instrumentação Dinâmica da Quarkslab (<i>Quarkslab Dynamic Binary Instrumentation</i>)
<i>RDTSC</i>	Leitura do Contador de <i>Time-Stamp</i> (<i>Read Time-Stamp Counter</i>)
<i>RSL</i>	Revisão Sistemática da Literatura (<i>Systematic Literature Review</i>)
<i>RSMB</i>	Assinatura bruta da SMBIOS (<i>Raw SMBIOS Table signature</i>)
<i>RVA</i>	Endereço Virtual Relativo (<i>Relative Virtual Address</i>)
<i>RWX</i>	Permissões de Leitura, Escrita e Execução (<i>Read, Write, Execute</i>)
<i>SBI</i>	Instrumentação Binária Estática (<i>Static Binary Instrumentation</i>)
<i>SDK</i>	Kit de Desenvolvimento de Software (<i>Software Development Kit</i>)
<i>SMBIOS</i>	BIOS de Gerenciamento do Sistema (<i>System Management BIOS</i>)
<i>SMC</i>	Código Automodificante (<i>Self-Modifying Code</i>)
<i>SO</i>	Sistema Operacional
<i>TEB</i>	Bloco de Ambiente da Thread (<i>Thread Environment Block</i>)
<i>TLS</i>	Armazenamento Local de Thread (<i>Thread Local Storage</i>)
<i>UAC</i>	Controle de Conta de Usuário (<i>User Account Control</i>)
<i>UPX</i>	Empacotador Universal de Executáveis (<i>Ultimate Packer for eXecutables</i>)
<i>VM</i>	Máquina Virtual (<i>Virtual Machine</i>)
<i>VMI</i>	Introspecção de Máquina Virtual (<i>Virtual Machine Introspection</i>)
<i>WMI</i>	Instrumentação de Gerenciamento do Windows (<i>Windows Management Instrumentation</i>)
<i>ZF</i>	Bandeira Zero (<i>Zero Flag</i>)

Sumário

1	Introdução	2
1.1	Motivação	3
1.2	Objetivos	4
1.3	Estrutura do Documento	4
2	Conceitos Básicos	5
2.1	Análise Estática	5
2.1.1	Desmontagem (<i>Disassembly</i>) e Decompilação	6
2.2	Análise Dinâmica	7
2.2.1	Depuração	7
2.2.2	Monitoramento do Processo em Execução	8
2.2.3	Instrumentação Binária Dinâmica (DBI)	8
2.2.3.1	Instrumentadores (DBI) para Windows	9
2.3	Técnicas de Proteção de Software	10
2.3.1	Ofuscadores	10
2.3.1.1	Alteração da Ordem do Código (<i>Change the Order of the Code</i>)	11
2.3.1.2	Inserção de Dados Redundantes (<i>Redundant Data Insertion</i>)	11
2.3.2	Packers	12
2.3.3	Ferramentas Especializadas de Proteção de Software	12
2.4	Overhead e Transparência	13
2.5	Considerações sobre o Capítulo	14
3	Revisão Sistemática da Literatura	15
3.1	Protocolo	15
3.1.1	Estratégia e Expressão de Busca	17
3.1.2	Critérios de Inclusão e Exclusão	18
3.1.3	Bases de Dados e Resultados da Busca	19
3.2	Revisão da Literatura	19
3.2.1	Técnicas de Evasão em Malware e Empacotadores/Protetores	20
3.2.2	Contramedidas e Ferramentas de Análise Baseadas em DBI Voltadas para Malware Evasivo	22
3.3	Discussão	26
4	Técnicas Evasivas	28
4.1	Anti-Disassembly	28
4.1.1	Instruções de Salto com o Mesmo Alvo	29
4.1.2	Instruções com Condição Constante	29

4.1.3	Mudança de Fluxo de Controle do Programa (Program Control Flow Change)	30
4.1.4	Ofuscação de APIs	31
4.1.5	Desafios para a Detecção Usando o Instrumentador Pin	31
4.2	Anti-Debug	32
4.2.1	PEB.BeingDebugged Flag (Função IsDebuggerPresent)	32
4.2.2	PEB.NtGlobalFlag	33
4.2.3	FindWindow	34
4.2.4	Técnicas de Medição de Tempo	35
4.2.4.1	Função GetTickCount	35
4.2.4.2	Função QueryPerformanceCounter	35
4.2.4.3	Instrução RDTSC	36
4.2.4.4	Outras Funções de Tempo	36
4.3	Anti-VM	36
4.3.1	Detecção Baseada em Registro (Registry-based Detection)	37
4.3.2	Detecção Baseada em Hardware (Hardware-based Detection)	39
4.3.3	Detecção de Processos e Serviços	40
4.3.4	Detecção Baseada em Sistema de Arquivos e <i>Drivers</i> no Sistema	40
4.4	Anti-Instrumentação	41
4.4.1	Detecção de Artefatos de Ambiente (<i>Environment Artifacts</i>)	42
4.4.2	Baseada em Exceções (<i>Exception-based</i>)	45
4.4.3	Conteúdo e Permissões de Memória (<i>Memory Contents and Permissions</i>)	46
4.4.4	Medidas Temporais (<i>Time Overhead</i>)	47
4.4.5	Outras Técnicas	48
4.5	Discussão	49
5	Contradef	50
5.1	Visão Geral da Ferramenta	50
5.2	Implementação da Contradef	51
5.3	Arquitetura	51
5.3.1	Análise Adaptativa com Detecção de Ferramentas de Proteção	52
5.3.1.1	Regra YARA para o Obsidium	53
5.3.1.2	Regra YARA para o Themida	55
5.3.2	Módulos da <i>Contradef</i>	56
5.3.2.1	Interceptação de Funções Críticas - <i>FunctionInterceptor</i>	56
5.3.2.2	Trace de Funções Externas - <i>TraceFcnCall</i>	57
5.3.2.3	Rastreamento de Memória - <i>TraceMemory</i>	59
5.3.2.4	Rastreamento de Instruções - <i>TraceInstructions</i>	60
5.3.2.5	Desmontagem Dinâmica do Código Assembly - <i>TraceDisassembly</i>	60
5.3.2.6	Analizador de sequência de código executado - <i>InstructionSequenceDetector</i>	61
5.3.2.7	Contramedidas para técnicas <i>anti-DBI</i> e <i>anti-VM</i>	63
5.4	Limitações	64
5.5	Discussão	65

6	Experimentações e Análises	67
6.1	Prova de Conceito	67
6.1.1	Ambiente de Experimentação	68
6.1.2	Resultados da Execução com Amostras de Teste	69
6.1.3	Execução do <i>Obsidium</i>	70
6.1.4	Execução do <i>Themida</i>	70
6.1.5	Limitações práticas e recomendações operacionais	72
6.1.6	Discussão	72
7	Análise com Amostras de Malwares	74
7.0.1	Ambiente de Experimentação	74
7.0.1.1	Seleção das Amostras	74
7.0.1.2	Máquina virtual	75
7.0.1.3	Condições de execução	75
7.0.2	Execução Monitorada	75
7.0.3	Estrutura da Apresentação dos Resultados	76
7.1	Técnicas Evasivas Encontradas no <i>Obsidium</i>	76
7.1.1	Técnica anti- <i>debug</i> : detecção de padrões nos títulos de janelas via <i>EnumWindows</i>	76
7.1.2	Técnica Anti-análise - Detecção de Overhead com a Função <i>GetTickCount</i>	78
7.1.3	Técnica Anti- <i>debug</i> - Detecção de Depuradores com a Função <i>RaiseException</i>	81
7.1.4	Técnica Anti- <i>debug</i> - Detecção do <i>TitanHide</i> (Ferramenta de Ocultação de Depuradores) com a Função <i>CreateFileW</i>	84
7.1.5	Técnica Anti- <i>Debug</i> – Detecção de Depuradores com a Função <i>NtQueryInformationProcess</i> e <i>ProcessDebugPort</i>	85
7.1.6	Técnica Anti- <i>Debug</i> – Detecção de Depuradores com a Função <i>NtQueryInformationProcess</i> e <i>ProcessDebugObjectHandle</i>	87
7.1.7	Técnica Anti- <i>DBI/Debug</i> – Detecção de Depuradores com a Função <i>NtQueryInformationProcess</i> e <i>ProcessDebugFlags</i>	88
7.1.8	Técnica Anti- <i>debug</i> - Detecção de Depuradores com a Função <i>CheckRemoteDebuggerPresent</i>	90
7.2	Técnicas Evasivas Encontradas no <i>Themida</i>	91
7.2.1	Técnica anti- <i>VM</i> : inspeção do SMBIOS via <i>GetSystemFirmwareTable</i> (RSMB)	91
7.2.2	Técnica anti- <i>VM</i> : consulta a chaves do Registro e artefatos ACPI/BIOS	93
7.2.3	Técnica anti-análise: verificação de nomes/títulos de janelas via <i>FindWindowA</i>	94
7.2.4	Técnica anti- <i>sandbox</i> : sondagem de DLLs via <i>GetModuleHandleExW</i>	96
7.2.5	Técnica anti-análise: enumeração de módulos do sistema via <i>RtlGetNativeSystemInformation</i> com varredura de padrões de texto	97
7.2.6	Outras técnicas anti- <i>debug</i> observadas no <i>Themida</i>	99
7.2.6.1	Detecção via <i>NtQueryInformationProcess/ProcessDebugPort</i>	99
7.2.6.2	Detecção via <i>NtQueryInformationProcess/ProcessDebugObjectHandle</i>	100
7.2.6.3	Detecção via <i>CheckRemoteDebuggerPresent</i>	100
7.3	Resumo das Técnicas Evasivas Encontradas	101
7.3.1	Discussão	101

8	Conclusões, Contribuições e Trabalhos Futuros	104
8.1	Considerações Finais	104
8.2	Contribuições Alcançadas	106
8.3	Trabalhos Futuros	106
	Referências Bibliográficas	108

Capítulo 1

Introdução

A evolução contínua do ecossistema de *malware* tem imposto desafios crescentes às técnicas tradicionais de análise. Softwares maliciosos direcionados ao sistema operacional Windows, sobretudo no formato *Portable Executable* (PE), permanecem entre os mais observados no campo forense e industrial ([Muralidharan et al., 2022](#)). Assim, a análise de *malwares* continua sendo uma atividade crucial para proteger sistemas e dados contra ataques maliciosos, envolvendo o processo de examinar detalhadamente o artefato para compreender seu funcionamento, identificar seus métodos e, conseqüentemente, definir formas de neutralizá-lo ou removê-lo ([Sikorski and Honig, 2012](#)).

No entanto, os criadores de *malware* frequentemente empregam técnicas avançadas de evasão, cujo objetivo é atrasar ou impedir a análise de executáveis por meio de observação comportamental e/ou engenharia reversa. Essas técnicas aumentam a vida útil do código malicioso em redes internas ou na Internet, bem como a anonimidade dos atacantes e de seus servidores de controle ([Hendrikse and Austen, 2012](#)). Estudos, como o de [Afianian et al. \(2018\)](#), indicam que *malwares* modernos utilizam técnicas avançadas de evasão e camuflagem para evitar a detecção por antivírus e dificultar a engenharia reversa. Um exemplo recorrente é o uso de empacotadores ou *packers*, que ocultam o código malicioso sob camadas de compactação, criptografia ou ofuscação, reduzindo a efetividade da análise estática ([Coccia et al., 2022](#)).

Além dos *packers*, autores de códigos maliciosos também podem recorrer a ferramentas comerciais de proteção de *software*, conhecidas genericamente como protetores. Embora tenham sido desenvolvidas originalmente para salvaguardar propriedade intelectual e dificultar a pirataria, tais ferramentas podem ser desviadas de seu propósito legítimo para aumentar a complexidade da análise forense e da detecção automatizada ([Lee et al., 2021](#)). Assim, para uma parcela relevante das amostras atuais, não basta apenas “executar e observar”: é preciso executar, observar e neutralizar sondagens do binário sobre o próprio ambiente de análise.

Dessa forma, métodos que dependem de heurísticas para detectar o término do processo de desempacotamento e despejar o conteúdo da memória comumente recorrem à execução dinâmica da amostra, por exemplo, com emuladores, depuradores ou instrumentadores ([Coccia et al., 2022](#)). Nesse cenário, os instrumentadores binários dinâmicos (*Dynamic Binary Instrumentation* – DBI) apresentam capacidades significativas para a análise de *malwares*, ao oferecer mecanismos que permitem inspecionar, acompanhar e, em alguns casos, manipular a execução em tempo real, mesmo diante de técnicas de camuflagem ou evasão.

Diferentemente da análise estática, diretamente afetada por empacotamento e ofuscação, a DBI permite acompanhar a execução real do binário, inspecionando chamadas de API, acessos à

memória, instruções críticas e regiões reconstruídas em tempo de execução. Além disso, diferentes linhas de pesquisa mostram que a DBI pode ser utilizada não apenas para *detectar* evasão, mas também para *mitigá-la* em tempo de execução, devolvendo ao binário respostas compatíveis com um ambiente nativo e permitindo que o comportamento de interesse se manifeste.

Com base nessas evidências, o objetivo deste trabalho é propor e avaliar uma estratégia de análise dinâmica voltada à **identificação** e **mitigação** de técnicas evasivas em executáveis Windows, utilizando o instrumentador binário dinâmico *Intel Pin*. Em particular, busca-se explorar, identificar e analisar técnicas evasivas, sejam elas anti-*disassembly*, anti-depuração, anti-virtualização ou anti-instrumentação, utilizadas por *malwares* para evitar a análise, bem como identificar os protetores empregados para dificultar sua inspeção.

Este trabalho se alinha a uma tendência apontada na literatura recente: a de que não há uma única técnica capaz de tornar o ambiente totalmente invisível ao *malware*, mas sim um conjunto de camadas que precisa ser continuamente atualizado à medida que novos protetores de *software*, novas versões e novas formas de detecção de DBI surgem. Assim, além de apresentar a ferramenta *Contradef* e sua arquitetura de instrumentação, esta dissertação busca contribuir com uma caracterização prática do problema, aproximando o cenário acadêmico dos casos reais em que o analista precisa *sobreviver* às rotinas de evasão antes de, de fato, analisar o código.

1.1 Motivação

A motivação deste trabalho decorre do avanço contínuo de técnicas de evasão que visam frustrar a observação e a engenharia reversa de executáveis maliciosos. *Malwares* cientes de contexto detectam quando estão sob análise — em máquinas virtuais, depuradores, *sandboxes* ou instrumentadores — e, ao identificar o ambiente, interrompem a execução, alteram o fluxo de controle ou postergam a ativação de cargas úteis (Balzarotti et al., 2010; Kirat et al., 2011; Kruegel, 2014; D’Elia et al., 2019; Soriano-Salvador and Guardiola-Múzquiz, 2023). Essa capacidade pode ser introduzida no *software* malicioso por meio de ferramentas de proteção, como *Themida* e *Obfuscator*, que combinam empacotamento, ofuscação e rotinas *anti-analysis*, convertendo o próprio mecanismo de proteção em vetor ativo de detecção de ambiente. A evasão estende-se, ainda, a sondagens de *hardware* — por exemplo, consultas a recursos da *CPU*, dispositivos e estrutura de memória — exigindo que o ambiente de análise replique ou mascare características específicas para evitar exposição de artefatos (Afianian et al., 2018; Gozzini et al., 2022). Em paralelo, verificações temporais e de desempenho (*timing checks*) ampliam o desafio, pois qualquer sobrecarga (*overhead*) ou inconsistência sutil pode denunciar a presença do observador.

Nesse contexto, abordagens puramente estáticas mostram-se insuficientes frente a empacotamento e virtualização de código, ao passo que a análise dinâmica precisa ser capaz de observar e sustentar a execução sem revelar artefatos do próprio ambiente. Os instrumentadores binários dinâmicos (DBI) surgem como alternativa promissora por permitirem inspecionar, e — quando necessário — alterar respostas do sistema em tempo de execução, aumentando a chance de observação do comportamento real do binário.

Diante desse cenário, torna-se necessário adotar abordagens de análise que não apenas observem o comportamento, mas que sejam capazes de acompanhar e intervir exatamente no momento em que o binário tenta sondar o ambiente. DBI é particularmente adequada porque atua no mesmo plano em que muitas dessas técnicas evasivas se manifestam — chamadas de API, instruções específicas, acessos à memória, disparo de exceções — permitindo registrar o evento evasivo, caracterizá-lo e, quando conveniente, *mascará-lo* em tempo de execução. A proposta deste trabalho nasce, portanto, de uma necessidade prática: analisar amostras protegidas ou empacotadas e neutralizar, de forma seletiva, os mecanismos de proteção empregados por

ferramentas como *Themida* e *Obsidium*, apenas o suficiente para que o comportamento real venha à tona. Ao mesmo tempo, pretende-se registrar esse comportamento em alto nível de detalhe, produzindo um panorama de execução que permita descrever, em nível de API, instrução e memória, como as técnicas evasivas se materializam. Ao priorizar uma análise dinâmica centrada no DBI e nas próprias técnicas de evasão — em vez de apenas listar contramedidas — este trabalho busca aproximar a investigação acadêmica das demandas do analista em campo: primeiro sobreviver às rotinas de detecção de ambiente; depois, de fato, observar e compreender o comportamento malicioso.

1.2 Objetivos

O objetivo geral deste trabalho é propor, implementar e avaliar uma estratégia de análise dinâmica, baseada em instrumentação binária dinâmica (DBI) com o *Intel Pin*, capaz de **identificar**, **caracterizar** e **mitigar** técnicas de evasão empregadas por executáveis Windows (PE), incluindo aquelas introduzidas por protetores comerciais (como *Themida* e *Obsidium*) que visam dificultar ou inviabilizar a análise.

Para alcançar esse objetivo, estabelecem-se os seguintes objetivos específicos:

- Projetar e desenvolver uma ferramenta de instrumentação baseada no *Intel Pin* (*Contradef*) capaz de registrar eventos associados a técnicas evasivas, preservando o máximo possível a transparência do ambiente;
- Desenvolver, mecanismos para identificar ferramentas de proteção, como *Themida* e *Obsidium*, otimizando a detecção e neutralização de técnicas evasivas;
- Implementar, na ferramenta, mecanismos de resposta/neutralização que permitam contornar sondagens do binário sobre o ambiente de análise e, assim, possibilitar que o comportamento malicioso se manifeste;
- Avaliar experimentalmente a ferramenta proposta com *malwares* reais protegidos e executados em ambiente de análise realista (VM), demonstrando a viabilidade prática da abordagem e apontando limitações.

1.3 Estrutura do Documento

Este trabalho está organizado em oito capítulos, incluindo esta introdução. No Capítulo 2 são apresentados os fundamentos teóricos e os conceitos necessários para a compreensão da proposta, com ênfase em análise de *malware*, técnicas de evasão e instrumentação binária dinâmica. O Capítulo 3 descreve a Revisão Sistemática da Literatura (RSL) conduzida, bem como os principais trabalhos relacionados que investigam evasão e transparência em ambientes de análise. No Capítulo 4 são detalhadas as técnicas evasivas mais frequentemente observadas em *malwares* e em protetores comerciais, organizadas por categoria (anti-*debug*, anti-VM, anti-instrumentação, entre outras). O Capítulo 5 apresenta a ferramenta de análise desenvolvida, denominada *Contradef*, sua arquitetura de instrumentação e os módulos responsáveis pela detecção e registro de comportamento evasivo. O Capítulo 6 descreve a prova de conceito (*proof of concept* – PoC) utilizada para validar a abordagem proposta. No Capítulo 7 são apresentados os experimentos realizados com amostras reais, executadas em ambiente de análise que replica o cenário utilizado por analistas (com uso de máquinas virtuais), bem como a discussão dos resultados. Por fim, o Capítulo 8 conclui a dissertação com as considerações finais, contribuições alcançadas e trabalhos futuros.

Capítulo 2

Conceitos Básicos

Neste capítulo, apresentamos os conceitos fundamentais necessários para a compreensão deste trabalho. Inicialmente, discutimos **análise estática** (desmontagem e decompilação) e **análise dinâmica** (depuração, monitoramento de processos e *DBI*), mostrando como cada abordagem oferece perspectivas distintas sobre o comportamento de *software* malicioso.

Em seguida, abordamos **técnicas de proteção de *software***, incluindo ofuscadores (alteração da ordem do código e inserção de dados redundantes), ***packers*** e **ferramentas comerciais de proteção**. Cada uma dessas técnicas visa dificultar o trabalho de analistas e soluções de segurança, seja camuflando o código ou impedindo a desmontagem e a depuração. Por fim, discutimos os aspectos de **overhead** e **transparência**, ressaltando como a introdução de instrumentação e monitoramento pode afetar o desempenho e a fidelidade da análise.

Cada um desses tópicos é explorado considerando suas interconexões e relevância para o tema central deste projeto. As definições básicas e explicações detalhadas são fornecidas para garantir uma compreensão clara e abrangente, necessária para o desenvolvimento e a compreensão do trabalho.

2.1 Análise Estática

A abordagem mais comum na detecção de *software* malicioso é a análise estática, que envolve examinar softwares sem executá-los (Mohanta and Saldanha, 2020), verificando o código e a estrutura da aplicação. É especialmente útil em situações onde a execução do *software* pode ser arriscada, como no caso de *malwares* altamente destrutivos ou quando o ambiente de execução não está disponível. Além disso, é uma boa alternativa quando se deseja compreender a lógica do *software*, realizar auditorias de segurança ou verificar conformidade com padrões de codificação sem o risco de ativar comportamentos maliciosos.

A análise estática pode envolver técnicas como a desmontagem (*disassembly*) e a decompilação. A desmontagem converte o código binário em uma representação de linguagem de montagem, enquanto a decompilação tenta reconstruir o código-fonte original em uma linguagem de alto nível. Existem ferramentas especializadas que analisam diretamente as estruturas dos arquivos executáveis, sendo capazes de inspecionar o código-fonte e identificar padrões nas diferentes seções dos executáveis. Essas ferramentas permitem uma análise detalhada e ajudam a entender a funcionalidade e identificar comportamentos maliciosos no *software* analisado.

Algumas ferramentas populares para análise estática incluem *IDA Free*¹, *Radare2*² e *Ghidra*³.

No entanto, também existem ferramentas, como os empacotadores (*packers*), que reduzem ou limitam a eficácia da análise estática, pois ocultam o código real do programa. Isso é feito compactando o código e frequentemente adicionando camadas de ofuscação, o que torna difícil para a análise estática interpretar a lógica e a funcionalidade do software. Além disso, muitos empacotadores incorporam técnicas *anti-análise* que detectam e impedem esforços de descompactação ou análise. Como a análise estática não executa o código, ela não pode acessar efetivamente o código descompactado ou descriptografado que só se revela durante a execução do programa.

Para estes cenários, existem ferramentas como *CAPA*⁴, *PEiD*⁵, *YARA*⁶ e *Detect It Easy (DIE)*⁷ que podem ser utilizadas na detecção de *packers*.

2.1.1 Desmontagem (*Disassembly*) e Decompilação

Quando desmontamos um executável, pegamos o binário como entrada e geramos o código em um formato mais legível para humanos em linguagem *assembly*, que podemos então ler para entender a funcionalidade e a intenção do executável que estamos analisando por engenharia reversa (Mohanta and Saldanha, 2020). Esse processo é feito geralmente com um desmontador (*disassembler*). Ferramentas comuns para essa finalidade incluem *IDA Free*⁸, *Binary Ninja*⁹, *BORG Disassembler*¹⁰ e *OllyDbg*¹¹.

A desmontagem de um executável envolve carregar o executável em um *disassembler* e analisar as instruções do programa para entender sua funcionalidade. As instruções executadas pela CPU podem revelar precisamente o comportamento do programa. Contudo, essa técnica exige conhecimentos especializados sobre o funcionamento e a arquitetura da CPU, a construção de códigos em linguagem *assembly* e os conceitos do sistema operacional Windows.

No entanto, a desmontagem enfrenta dificuldades com *malwares* que utilizam técnicas para esconder seu código e intenções, tornando o código desmontado complexo e difícil de interpretar. A eficácia da desmontagem depende significativamente da habilidade do analista em interpretar o código desmontado, o que pode ser desafiador diante de técnicas complexas e inovadoras usadas pelos *malwares* como a ofuscação ou empacotamento, conceitos que trataremos mais adiante.

Embora um processo de desmontagem possa parecer simples, existem diversas técnicas que os atacantes usam para enganar o programa desmontador e fazê-lo produzir um código *disassembly* diferente do que está sendo executado de fato (Or-Meir et al., 2019).

Já os descompiladores são extremamente úteis para determinar a lógica de um programa em uma linguagem de alto nível. No entanto, a decompilação, embora útil na análise de software, pode não ser muito eficaz na análise de *malwares* devido à sua natureza complexa e às técnicas avançadas frequentemente empregadas pelos *malwares*.

Durante o processo de compilação, devido às diferenças entre o código de uma linguagem de alto nível e o código de máquina, informações como símbolos e comentários, que são cruciais

¹<https://hex-rays.com/ida-free>

²<https://rada.re/r/down.html>

³<https://ghidra-sre.org/>

⁴<https://mandiant.github.io/capa/>

⁵<https://github.com/wolfram77web/app-peid>

⁶<https://github.com/virustotal/yara/releases>

⁷<https://github.com/horsicq/Detect-It-Easy>

⁸<https://hex-rays.com/ida-free>

⁹<https://binary.ninja/free/>

¹⁰<https://github.com/opsxcq/disassembler-borg>

¹¹<https://www.ollydbg.de/download.htm>

para a compreensão do código-fonte, são removidas (Cao et al., 2022). Essa remoção também dificulta a interpretação do código quando ele é decompilado. A maioria dos sistemas de decompilação não consegue converter diretamente programas em 100% de código-fonte. Em vez disso, eles geralmente fornecem uma representação intermediária próxima da original (Hoglund and McGraw, 2003).

Portanto, a eficácia da decompilação está intimamente ligada ao contexto específico do código, incluindo a linguagem de programação e as técnicas de compilação utilizadas, que podem variar amplamente adicionando uma camada extra de complexidade à análise.

2.2 Análise Dinâmica

No contexto da análise de *software* malicioso, a análise dinâmica consiste em executar uma amostra utilizando o auxílio de diversas ferramentas e registrar não apenas o comportamento, mas também observar os vários artefatos gerados pelo *malware* executado (Mohanta and Saldanha, 2020). Esta abordagem envolve a execução do software em um ambiente controlado e monitorado para observar seu comportamento em tempo real. Isso permite que os analistas vejam como o *malware* se comporta durante a execução, detectando atividades maliciosas que podem não ser evidentes na análise estática.

Essa abordagem é particularmente vantajosa para monitoramento de chamadas de função, análise de parâmetros de função, rastreamento de instruções e acompanhamento do fluxo de informações. Tais técnicas são frequentemente empregadas para detectar comportamentos maliciosos, como tentativas de conexão com servidores de comando e controle (*command and control*), modificações no sistema e outras atividades suspeitas. Ao executar o *malware*, é possível observar diretamente suas ações. Além disso, a análise dinâmica pode ajudar a superar as técnicas de ocultação de código, pois, com a execução, as camadas de proteção do código podem ser removidas, revelando o comportamento real do *malware*. Outro benefício importante é a identificação de técnicas *anti-análise*, pois o comportamento do *malware* durante a execução pode revelar a presença de tais técnicas, que não são detectáveis pela análise estática.

Normalmente, essa análise é realizada após a análise estática básica ter atingido um impasse, seja devido a alguma técnica *anti-análise* ou ao esgotamento das técnicas de análise estática disponíveis. Ela pode envolver o monitoramento do *malware* enquanto ele é executado ou a análise do sistema após a execução do *malware*. Dentre as ferramentas que facilitam o processo de análise dinâmica de *malware*, podemos destacar *ProcMon*¹², *Wireshark*¹³, *OllyDbg*¹⁴ e *Cuckoo Sandbox*¹⁵.

A aplicação prática da análise dinâmica na detecção de *malware* pode ser dividida em três etapas: depuração, que permite aos analistas acompanhar a execução do programa passo a passo; monitoramento do processo em execução, que permite rastrear chamadas de função, eventos do sistema e comunicação em tempo real; e o emprego de instrumentadores DBI, para observar detalhadamente o fluxo de instruções e a introdução de lógicas personalizadas para detectar e mitigar atividades maliciosas durante a execução.

2.2.1 Depuração

Os depuradores permitem que desenvolvedores e analistas de *malware* monitorem e controlem a execução de programas (Mohanta and Saldanha, 2020), identificando e corrigindo erros em

¹²<https://learn.microsoft.com/pt-br/sysinternals/downloads/procmon>

¹³<https://www.wireshark.org/download.html>

¹⁴<https://www.ollydbg.de/download.htm>

¹⁵<https://cuckoosandbox.org/download.html>

tempo real, sendo programas amplamente utilizados no processo de desenvolvimento de software. Durante esse processo, o desenvolvedor pode visualizar os dados dos símbolos ou variáveis carregadas em memória e também acompanhar o fluxo de execução. Em suma, essas ferramentas são essenciais no desenvolvimento e análise de software, pois proporcionam insights valiosos sobre a execução do código que não podem ser obtidos com desassembladores.

Existem duas maneiras principais de depurar um programa. A primeira é iniciar o programa com o depurador. Quando o programa é carregado na memória, ele para imediatamente antes da execução de seu ponto de entrada, permitindo controle total do programa. A outra maneira é anexar um depurador a um programa já em execução. Nesse caso, todas as *threads* do programa são pausadas, possibilitando a depuração (Sikorski and Honig, 2012). Ambas as formas de depurar um programa oferecem flexibilidade na análise, especialmente em cenários complexos envolvendo *malware*.

No contexto da análise de *malware*, os depuradores são particularmente úteis para observar o comportamento durante a execução e detectar técnicas *anti-análise* utilizadas para evitar a detecção. Sua capacidade de manipular exceções permite um controle refinado sobre o fluxo de execução, favorável para desvendar comportamentos maliciosos ocultos. No entanto, essa mesma capacidade pode ser explorada por *malwares* para evadir a análise, utilizando, por exemplo, técnicas *anti-debugging*, o que demonstra a necessidade de uma compreensão profunda das técnicas de depuração e de evasão.

2.2.2 Monitoramento do Processo em Execução

O monitoramento do processo em execução é uma técnica importante na análise dinâmica de software, especialmente útil na análise de *malwares*. Existem várias ferramentas que desempenham as funções de monitoramento de um processo como *NtTrace*¹⁶, *Wireshark*¹⁷, *Cuckoo Sandbox*¹⁸ e *ProcMon*¹⁹.

Tomando a *ProcMon*, ferramenta de monitoramento avançada para Windows, como exemplo, percebemos que é possível interceptar e registrar eventos do sistema em tempo real, como leitura e escrita em arquivos, acesso ao registro, criação e término de processos e *threads*, e comunicação com a rede. É possível configurar filtros para focar em eventos específicos, como atividades de um processo em particular, tipos específicos de acesso ao registro ou a arquivos, entre outros. Isso permite que os analistas capturem dados das atividades mais relevantes, facilitando a análise e permitindo observar o comportamento do processo em tempo real, identificando atividades suspeitas ou maliciosas.

Na análise de *malware*, o *ProcMon* é utilizado para detectar comportamentos maliciosos, como tentativas de conexão com servidores de comando e controle (*command and control*), modificação de arquivos do sistema ou acesso não autorizado ao registro. Ele também ajuda a identificar técnicas de evasão, monitorando como o *malware* interage com o ambiente e detectando verificações de ambiente virtual ou tentativas de desativar software de segurança. Além disso, o *ProcMon* permite avaliar as mudanças feitas pelo *malware* no sistema, incluindo a criação de novos arquivos, alterações no registro e comunicação de rede.

2.2.3 Instrumentação Binária Dinâmica (DBI)

Instrumentação Binária Dinâmica (*Dynamic Binary Instrumentation* – DBI) é uma técnica que permite a inspeção e inserção de código em um programa binário durante sua execução,

¹⁶<https://github.com/rogerorr/NtTrace/releases>

¹⁷<https://www.wireshark.org/download.html>

¹⁸<https://cuckoosandbox.org/download.html>

¹⁹<https://learn.microsoft.com/pt-br/sysinternals/downloads/procmon>

viabilizando a análise e modificação de seu comportamento sem a necessidade de acesso ao código-fonte original. Segundo [Rodríguez et al. \(2014\)](#), DBI é especialmente vantajosa por não exigir a recompilação do software, proporcionando independência em relação à linguagem de programação e permitindo a análise detalhada de executáveis e de código gerado dinamicamente.

DBI é amplamente utilizada em diferentes áreas, incluindo análise de segurança, depuração e otimização de desempenho de software, pois possibilita a observação detalhada das operações de um programa em tempo real. [D'Elia et al. \(2019\)](#) descrevem a DBI como um paradigma que insere artefatos de análise em pontos específicos de um programa enquanto ele está em execução, sem a necessidade de manipulação direta do código binário. Esse processo é facilitado por frameworks como *Pin* e *DynamoRIO*, que abstraem as particularidades da arquitetura de hardware e fornecem uma interface de instrumentação para os desenvolvedores.

Ao injetar código de instrumentação dentro do binário em análise, as ferramentas DBI fornecem um controle profundo sobre o programa analisado, oferecendo uma visão tanto de baixo nível (instruções) quanto de alto nível (chamadas de sistema) ([Polino et al., 2017a](#)). Ferramentas de DBI também são úteis para o monitoramento de aplicativos proprietários e de sistemas operacionais como o Windows, onde o código-fonte normalmente não está disponível e a interface do *kernel* não pode ser adaptada para observabilidade ([Skaletsky et al., 2010](#)). Dessa forma, a instrumentação de binários não modificados é valiosa para análises de desempenho e monitoramento em cenários realistas, sem interferir no comportamento geral do sistema. Assim, a DBI pode ser utilizada em contextos comerciais e de segurança, onde frequentemente o binário é a única forma disponível do programa ([Bernat et al., 2011](#)).

[Luk et al. \(2005\)](#) e [Bruening et al. \(2012\)](#) ampliam essa definição ao abordar a instrumentação como um método de virtualização de processos, onde o código original do aplicativo e o código inserido pela ferramenta de análise são intercalados dinamicamente durante a execução. Esse intercalamento permite que a DBI ofereça controle granular sobre as instruções e dados em execução, possibilitando a coleta de informações em tempo real e o monitoramento de comportamento.

2.2.3.1 Instrumentadores (DBI) para Windows

A seguir, apresentamos alguns dos instrumentadores mais conhecidos para ambientes Windows:

- *Frida*: framework DBI que permite a instrumentação e manipulação de aplicativos em tempo de execução ([Frida Docs, 2024](#)). Permite injetar trechos de *JavaScript* ou uma biblioteca própria em aplicativos nativos de diversos sistemas, incluindo Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD e QNX ([Frida Docs, 2024](#)). Também fornece algumas ferramentas simples construídas sobre a *API* do próprio Frida.
- *QBDI* (*Quarkslab Dynamic Binary Instrumentation*): framework DBI modular, multiplataforma e multi-arquitetura ([Quarkslab, 2024](#)), que permite que ferramentas de instrumentação baseadas nele sejam compiladas como bibliotecas dinâmicas e possam ser carregadas no processo-alvo com qualquer técnica de injeção. O QBDI é totalmente compatível com o Frida, permitindo a combinação das capacidades de ambas as ferramentas.
- *DynamoRIO*: sistema de manipulação de código em tempo de execução que suporta transformações de código em qualquer parte de um programa enquanto ele é executado ([DynamoRIO, 2024](#)). Inclui um framework de instrumentação dinâmica de código aberto, mantido pela *VMware*²⁰, contendo um conjunto de ferramentas e uma biblioteca que permitem a modificação de código em tempo de execução para finalidades como análise

²⁰<https://www.vmware.com/>

de software, otimização e segurança. Permite modificações arbitrárias nas instruções de aplicativos por meio de uma poderosa biblioteca de manipulação de instruções, permitindo manipulação eficiente e abrangente de aplicativos nativos ([DynamoRIO, 2024](#)).

- *DynInst*: é uma *API* de instrumentação dinâmica de código aberto que permite a modificação de código em tempo de execução, viabilizando a análise e adaptação de programas ([Paradyn, 2008](#)). Também permite alterar ou remover chamadas de sub-rotinas do programa. Foi projetada para que um único processo de instrumentação possa inserir trechos de código em vários processos executando em uma única máquina. Para suportar múltiplos processos, duas abstrações adicionais, processos e imagens (representação estática de um programa no disco), são incluídas na *API* ([Paradyn, 2008](#)).
- *Intel Pin*: é uma ferramenta de instrumentação binária dinâmica que permite a criação de plug-ins ou ferramentas de análise chamadas de *Pintools*, que podem ser usadas para análise de programas. Oferece suporte a múltiplas arquiteturas, permitindo a instrumentação em tempo de execução de arquivos binários compilados, sem a necessidade de recompilar o código-fonte. Isso é especialmente útil para a engenharia reversa e análise de *malwares* evasivos, pois permite que os analistas observem o comportamento real do *malware* em tempo de execução ([Intel, 2024](#)). O Pin opera como um compilador *just-in-time* (JIT), interceptando a execução da primeira instrução do executável e gerando novo código para a sequência de instruções a partir desse ponto. O código original é usado apenas como referência, e o Pin permite a injeção de código personalizado (instrumentação) ([Intel, 2024](#)).

Dentre todas as ferramentas descritas, Pin é extremamente poderosa para a engenharia reversa e análise de *malwares*. Sua capacidade de instrumentação dinâmica em tempo de execução, aliada à flexibilidade dos *Pintools*, permite uma análise detalhada e adaptativa, superando muitas das limitações das abordagens de análise dinâmicas tradicionais, como a depuração de código. Portanto, utilizaremos o Pin como a ferramenta principal para alcançar os objetivos deste trabalho.

2.3 Técnicas de Proteção de Software

As técnicas de proteção de software englobam métodos e ferramentas desenvolvidos para salvar aplicações contra engenharia reversa, cópia ilegal e modificação não autorizada. Entre as principais abordagens destacam-se a ofuscação de código — que altera a estrutura do programa para elevar consideravelmente a complexidade de sua compreensão lógica — e os *packers*, que comprimem e modificam a estrutura de um executável. Essas técnicas estabelecem camadas de proteção que visam exaurir o esforço de análise e a interpretação do código por parte de atacantes ou pesquisadores de segurança. Adicionalmente, protetores comerciais, como o *Themida* e o *Obsidium*, combinam múltiplas estratégias para assegurar que a lógica original permaneça oculta, ampliando significativamente os obstáculos à análise e à mitigação dos sistemas de proteção.

2.3.1 Ofuscadores

A ofuscação de código visa incrementar a complexidade intelectual do software, ocultando algoritmos, estruturas de dados e a lógica original para salvaguardar a propriedade intelectual contra a engenharia reversa não autorizada ([Behera and Bhaskari, 2015](#)). O propósito fundamental desse mecanismo é elevar o esforço necessário para a reconstrução da lógica do programa sem, contudo, tornar as instruções estritamente inacessíveis. Dessa forma, embora tais técnicas não

impeçam a desmontagem (*disassembly*) do executável, elas representam um desafio crítico à interpretação correta do comportamento do código.

É importante ressaltar que, embora a ofuscação possa ser aplicada tanto ao código-fonte — como observado em linguagens de *script* a exemplo do *JavaScript* — quanto a arquivos executáveis, o escopo deste trabalho concentra-se primordialmente na ofuscação em nível de binário. Essa distinção é relevante, pois a manipulação direta de instruções de máquina e da estrutura do executável impõe desafios singulares à análise dinâmica e à reconstrução do fluxo de controle original, diferenciando-se dos métodos aplicados a linguagens de alto nível.

Nesse contexto, as técnicas aplicadas ao binário adicionam camadas de opacidade que transformam rotinas funcionais em estruturas densas e complexas. Dentre os métodos frequentemente adotados para dificultar a análise de *malware*, destacam-se a alteração de blocos de código e a inserção de dados redundantes, táticas que visam dispersar o fluxo de execução para dificultar a compreensão lógica da aplicação.

2.3.1.1 Alteração da Ordem do Código (*Change the Order of the Code*)

Essa técnica simples de ofuscação altera a ordem de execução das instruções do programa. Ao mudar a sequência do código, o *malware* se torna mais difícil de analisar, pois a estrutura original é rearranjada, confundindo os analistas que tentam entender o fluxo lógico do código (Singh and Singh, 2018).

1 : push eax	1 : dec esi
2 : dec esi	2 : push eax
3 : add [eax],al	3 : jmp 7
4 : or al,[eax]	4 : cmp al,[eax]
5 : add [eax],al	5 : pop esp
6 : push 0x0	6 : jmp 12
7 : cmp al,[eax]	7 : add [eax],al
8 : pop esp	8 : push 0x0
9 : add bl,dh	9 : or al,[eax]
10 : xchg edi,eax	10 : add [eax],al
	11 : jmp 4
	12 : xchg edi,eax
	13 : add bl,dh

(a) (b)

Figura 2.1: (a) Código *assembly* x86 original, (b) Código reordenado usando instrução de salto incondicional. Singh and Singh (2018)

Na Figura 2.1, observamos que embora seja simples identificar a ordem original neste exemplo, para centenas de linhas de código, torna-se complicado para o analista descobrir a sequência real.

2.3.1.2 Inserção de Dados Redundantes (*Redundant Data Insertion*)

Essa técnica afeta principalmente a análise estática, inserindo dados redundantes ou “código morto” no programa. Esse código nunca é executado, mas está presente para confundir a análise estática, tornando difícil distinguir entre o código relevante e o irrelevante Singh and Singh (2018).

O código redundante ou “código morto” não afeta a finalidade original do software malicioso. Instruções de salto incondicionais são usadas para ignorar o bloco de código redundante,

```

1 : push eax
2 : nop           // Not Operation
3 : dec esi
4 : add [eax],al
5 : nop
6 : or al,[eax]
7 : jmp 12
8 : mov [ebp-0x18],esp
9 : mov [ebp-0x14],0x4010e0 // Redundant Code
10 : and eax,0x1
11 : mov [ebp-0x10],eax
12 : add [eax],al
13 : push 0x0
14 : cmp al,[eax]
15 : nop
16 : pop esp

```

Figura 2.2: Inserção de código redundante (código lixo). Singh and Singh (2018)

mantendo a ordem de execução original do *malware*. Além disso, uma série de instruções *NOP* (não operação, como instrução em x86) é inserida no *malware* para criar uma nova variante, conforme mostrado na Figura 2.2.

Essas técnicas demonstram como a ofuscação pode criar múltiplas camadas de complexidade para os analistas e sistemas de detecção automatizados, prolongando a eficácia dos *malwares*.

2.3.2 Packers

Ferramentas de empacotamento são usadas principalmente para reduzir o tamanho de um software e protegê-lo contra modificações não autorizadas. No entanto, quando aplicadas a *malwares*, essas ferramentas têm o propósito de ocultar o código malicioso, tornando sua análise e detecção mais difíceis (Coccia et al., 2022). O empacotamento de arquivos executáveis, que é uma das técnicas mais comuns para proteção de código, foi reaproveitado para ofuscação de código por autores de *malwares* como forma de burlar detectores de *malwares* (Muralidharan et al., 2022).

Quando um programa empacotado é executado, um pequeno subprograma ou *wrapper* é responsável por descompactar o conteúdo e, em seguida, executar o código descompactado. Durante a análise estática de um programa empacotado, apenas o subprograma pode ser analisado, o que limita a visibilidade do código real (Sikorski and Honig, 2012). Alguns empacotadores utilizam várias camadas de empacotamento, onde cada camada deve ser desempacotada para que a seguinte possa ser processada, complicando ainda mais a análise do *malware*. O *UPX*²¹ é um dos empacotadores mais populares usados. Isso pode ser atribuído ao fato de ser gratuito e fácil de usar; além disso, o *UPX* não foi incluído em listas negras por fornecedores de antivírus (Muralidharan et al., 2022).

2.3.3 Ferramentas Especializadas de Proteção de Software

Os protetores de software são ferramentas desenvolvidas com o principal objetivo de proteger software legítimo contra ataques de *hackers*, salvaguardar informações sensíveis e proteger algoritmos proprietários. Esses protetores utilizam uma série de técnicas avançadas para dificultar

²¹<https://upx.github.io/>

a engenharia reversa e a análise do código, tornando a tarefa dos atacantes significativamente mais complexa.

Uma das principais motivações para o uso de protetores é a proteção contra ataques de *hackers*, que frequentemente tentam descompilar, modificar ou explorar vulnerabilidades em software para realizar atividades maliciosas, como a inserção de código malicioso, o roubo de informações sensíveis ou a quebra de sistemas de licenciamento. Ao empregar técnicas de ofuscação, criptografia de código e detecção de depuradores, esses protetores tornam a análise e modificação do software extremamente difícil, aumentando a segurança contra tais ataques.

Além de proteger contra ataques de *hackers*, protetores são utilizados para salvaguardar informações sensíveis que possam estar embutidas no software. Isso inclui dados pessoais, chaves de criptografia, informações de autenticação e outros dados confidenciais que, se expostos, poderiam causar sérios prejuízos. A proteção dessas informações é crítica para manter a integridade e a confidencialidade dos dados, além de cumprir regulamentações e políticas de privacidade.

Algumas ferramentas de proteção de software incluem: *Obsidium*²², *Themida*²³, *CodeVirtualizer*²⁴, *VMProtect*²⁵, *Agile.NET*²⁶, *ASProtect*²⁷

2.4 Overhead e Transparência

No atual cenário da detecção de *malwares*, existem dois aspectos relevantes quando falamos do processo de detecção e identificação: *overhead* e transparência.

O conceito de *overhead* refere-se ao custo adicional imposto no tempo de execução de um programa devido à inserção de instruções extras ou processos que não fazem parte da operação normal do software. Ferramentas que introduzem menos *overhead* são mais desejáveis para análises em tempo real, pois dificultam a detecção do ambiente de análise. Em ambientes DBI, esse *overhead* é gerado pela necessidade de monitorar continuamente o comportamento do programa e pela modificação dinâmica do código, o que resulta em um aumento significativo no tempo de execução, consumo de *CPU* e uso de memória (Lee et al., 2021; Polino et al., 2017a).

Arafa et al. (2013) detalham como o *overhead* gerado pelo *Pin*, um dos frameworks mais usados, pode variar de acordo com a complexidade do código e a frequência de execução das rotinas de instrumentação. Eles apresentam métricas precisas, mostrando que *pintools* como *memtrace* e *inscount* introduzem um *overhead* de até 4,74 vezes para certos tipos de *benchmarks*. Crăciun et al. (2023) complementam que esse aumento de sobrecarga pode se manifestar de várias maneiras, como crescimento no uso de memória devido ao cache estendido e aumento no número de contextos de execução.

Já o conceito de transparência em ambientes de análise de *malware* é abordado de diferentes maneiras, refletindo as complexidades envolvidas na tentativa de manter a invisibilidade das ferramentas de análise para não desencadear comportamentos evasivos.

Segundo Afianian et al. (2018), a transparência se refere à capacidade de um sistema de ocultar pistas e sinais que poderiam revelar sua presença, tornando o ambiente de análise indistinguível de um sistema de produção. O autor destaca que a transparência está diretamente relacionada à capacidade do *malware* de reconhecer características específicas do ambiente em que está sendo executado. Por exemplo, técnicas de *fingerprinting* são comumente usadas por

²²<https://www.obsidium.de/product/sps/download>

²³<https://www.oreans.com/download.php>

²⁴<https://www.oreans.com/download.php>

²⁵<https://vmpsoft.com/>

²⁶<https://secureteam.net/>

²⁷<https://www.aspack.com/>

malwares para identificar se estão em um ambiente de análise, como um *debugger* ou uma máquina virtual.

Singh and Singh (2018) complementam essa definição ao afirmar que a transparência é essencial para que a análise dinâmica não seja detectada pelo *malware*. Os autores destacam que a análise e o sistema de produção devem ser indistinguíveis entre si, sugerindo que ambientes como *bare-metal* — caracterizados pela execução do sistema operacional diretamente sobre o *hardware* físico, sem a intermediação de uma camada de virtualização — são mais eficazes em termos de transparência por serem menos vulneráveis a ataques temporais e de detecção de artefatos comuns em máquinas virtuais.

Por outro lado, Balzarotti et al. (2010) expandem o conceito de transparência ao apresentarem cinco requisitos para que um sistema seja considerado verdadeiramente transparente, sendo eles: executar com maior privilégio que o código analisado; evitar efeitos colaterais acessíveis ao *malware*; preservar a semântica das instruções; lidar de maneira transparente com exceções; e manter a mesma medida de tempo que um sistema real.

Esses requisitos são considerados difíceis de implementar de maneira consistente, uma vez que até mesmo pequenas diferenças no comportamento do sistema podem ser detectadas por técnicas evasivas mais sofisticadas.

2.5 Considerações sobre o Capítulo

Neste capítulo, abordamos um amplo conjunto de temas importantes para a compreensão da análise e proteção dos *malwares*. Inicialmente, discutimos aspectos sobre análise estática e análise dinâmica, detalhando os processos de desmontagem, decompilação, depuração e monitoramento do processo em execução, além de ressaltar a relevância da Instrumentação Binária Dinâmica (DBI) para observar o comportamento do software em tempo real.

Na sequência, debatemos as técnicas de proteção de software, incluindo ofuscadores, *packers* e ferramentas comerciais, fundamentais para compreender como *malwares* se ocultam e dificultam a engenharia reversa.

E, por último, avaliamos a questão do overhead e da transparência no contexto da análise de *malwares*. Os métodos discutidos, principalmente aqueles que envolvem instrumentação, podem impactar o desempenho e introduzir artefatos que influenciam a precisão dos resultados. Assim, considerar a redução de *overhead* e a otimização da transparência mostra-se crucial para a análise de *malwares* modernos, assegurando eficiência e menor suscetibilidade a técnicas de evasão. Dessa forma, o conhecimento adquirido sobre cada tópico ajuda a fortalecer a capacidade para identificar, analisar e mitigar ameaças avançadas.

Capítulo 3

Revisão Sistemática da Literatura

Este capítulo apresenta um panorama crítico da literatura em segurança da informação com foco na análise dinâmica de executáveis e nas técnicas empregadas para *evitar* ou *detectar* ambientes ou ferramentas de análise, principalmente as de instrumentação binária dinâmica (DBI). Os trabalhos são organizados a partir de três eixos: (i) instrumentação binária dinâmica e seus *frameworks*; (ii) técnicas de evasão, incluindo *anti-debug*, *anti-VM*, *anti-instrumentação* e *anti-disassembly*; (iii) métodos para identificar e mitigar técnicas evasivas empregadas por *malwares* para se ocultarem, bem como a detecção das ferramentas utilizadas para esse propósito.

Diferente de uma revisão narrativa tradicional, a presente análise é conduzida sob a forma de uma Revisão Sistemática da Literatura (RSL). Isso garante maior rigor metodológico, transparência na seleção dos estudos e reprodutibilidade dos resultados. A RSL busca não apenas mapear o estado da arte, mas também identificar limitações nas soluções existentes e apontar direções para pesquisas futuras.

A revisão sistemática que orienta este capítulo fornece a base conceitual e empírica para o desenvolvimento da ferramenta proposta nesta dissertação, permitindo identificar contramedidas com suas respectivas implementações práticas e direcionar esforços para desafios abertos. A seção seguinte descreve, em detalhe, o protocolo adotado na condução da RSL, incluindo objetivos, questões de pesquisa, estratégias de busca, critérios de inclusão e exclusão, e métodos de extração e síntese.

3.1 Protocolo

A Revisão Sistemática da Literatura (RSL) deste trabalho segue as diretrizes de Kitchenham ([Kitchenham and Charters, 2007](#)), amplamente adotadas em estudos baseados em evidências nas áreas de Engenharia de Software e Segurança da Informação. A aplicação desse protocolo assegura rigor metodológico, reprodutibilidade e transparência em todas as etapas de seleção, análise e síntese dos estudos incluídos.

Para alinhar metas, questões e indicadores de qualidade, empregamos o paradigma *Goal–Question–Metric* (GQM) ([Basili et al., 2010](#)), que permite estruturar a revisão a partir de objetivos claros, questões específicas de pesquisa e métricas associadas. O processo foi organizado em três macroetapas: (i) **planejamento** da revisão (definição de objetivo GQM, fontes, estratégia de busca e critérios de inclusão/exclusão); (ii) **execução** (busca, triagem por título/resumo, leitura integral e extração de dados); e (iii) **síntese** e interpretação (agregação dos resultados, avaliação de qualidade e identificação de lacunas).

Objetivo da RSL. Analisar a produção científica sobre técnicas de evasão contra ambientes e ferramentas de análise, com foco na análise dinâmica de executáveis Windows (PE) com instrumentação binária dinâmica (DBI) — com ênfase no *Intel Pin* e nos aspectos de *transparência/invisibilidade* dessa ferramenta — e a identificação/caracterização de técnicas evasivas (anti-desmontagem, anti-depuração, anti-virtualização e anti-instrumentação), seus *sinais observáveis* (APIs, instruções, efeitos temporais) e *impactos sobre DBI* (detecção, falhas, degradação), incluindo a detecção de protetores comerciais (e.g., *Themida*, *Obsidium*). Pretende-se caracterizar o estado da arte, mapear contramedidas e evidências de eficácia e identificar lacunas para orientar pesquisas futuras. A formalização do objetivo, segundo GQM, encontra-se na [Tabela 3.1](#).

Tabela 3.1: Objetivo da RSL conforme o paradigma GQM.

Analisar	Publicações científicas revisadas por pares (e, secundariamente, teses/dissertações com contribuição técnica verificável).
Com o propósito de	Caracterizar técnicas evasivas (anti-desmontagem, anti-depuração, anti-VM, anti-instrumentação), ferramentas e contramedidas baseadas em DBI com ênfase em <i>transparência/invisibilidade</i> e eficácia na presença de protetores (e.g., <i>Themida</i> , <i>Obsidium</i>).
Em relação a	Estado da arte; cobertura por categoria de evasão (tempo (<i>overhead</i>), ambiente/sistema, cache/JIT, exceções); métricas de avaliação; reprodutibilidade e disponibilidade de artefatos.
Do ponto de vista	Pesquisadores, analistas de <i>malware</i> e projetistas de ferramentas DBI.
No contexto	Análise dinâmica de executáveis <i>Windows</i> (PE) em x86/x64, usando DBI (com ênfase no <i>Intel Pin</i>), incluindo cenários com protetores e empacotadores.

Questões de pesquisa.

- **RQ1:** Quais técnicas evasivas (anti-*disassembly*, anti-*debug*, anti-VM, anti-instrumentação) são reportadas em binários Windows (PE)?
- **RQ2:** Quais ferramentas foram usadas para aplicar/implantar as técnicas (e.g., *Themida*, *Obsidium*), como essas técnicas se *materializam* (APIs, instruções, memória, tempo) e em que camadas atuam (SO/hardware), quais sinais produzem e quais componentes são afetados?
- **RQ3:** Quais *frameworks* DBI (PIN, DynamoRIO, Frida, QBDI, etc.) são empregados nos estudos?
- **RQ4:** Quais evidências empíricas demonstram a eficácia/limitações dos *frameworks* DBI frente às técnicas evasivas?
- **RQ5:** Quais lacunas e oportunidades de pesquisa permanecem abertas (categorias pouco cobertas, portabilidade, reprodutibilidade)?

Métricas de análise. Para responder às questões, foram extraídas métricas organizadas em cinco eixos: (i) catálogo de técnicas evasivas, (ii) caracterização técnica das técnicas evasivas, (iii) adoção de *frameworks* DBI, (iv) evidências empíricas e replicabilidade, e (v) lacunas identificadas. Essas métricas estão resumidas na [Tabela 3.2](#).

Tabela 3.2: Métricas de análise derivadas do paradigma GQM.

M1	Catálogo de técnicas evasivas: taxonomia por categoria (anti- <i>disassembly</i> , anti- <i>debug</i> , anti-VM, anti-instrumentação), granularidade (API, instrução, chamada de sistema) e escopo (processo, sistema).
M2	Caracterização técnica das técnicas evasivas: ferramentas implicadas (empacotadores, protetores, ofuscadores), primitivos usados (APIs, instruções, syscalls, memória/temporização), sinais observáveis (comportamentais/temporal), alvo afetado (DBI, depuradores, VMs).
M3	Adoção de <i>frameworks</i> DBI: quais <i>frameworks</i> são empregados (PIN, DynamoRIO, Frida, QBDI, etc.), versões/plataformas (x86/x64, Windows) e modo de instrumentação (JIT/cache).
M4	Evidências empíricas e replicabilidade: número/tipo de amostras, presença de protetores, disponibilidade de artefatos (código, scripts), replicabilidade e detalhamento de protocolo experimental.
M5	Lacunas: categorias pouco exploradas, limitações de desempenho/portabilidade, ausência de código/datasets reproduzíveis, ameaças à validade.

Traçabilidade RQ–Métricas. A [Tabela 3.3](#) apresenta a relação entre as questões de pesquisa e as métricas utilizadas, evidenciando a aplicação do paradigma GQM.

Tabela 3.3: Traçabilidade entre questões de pesquisa (RQ) e métricas (M).

Questão de Pesquisa	Métricas Associadas
RQ1	M1, M2
RQ2	M1, M2
RQ3	M3
RQ4	M3, M4
RQ5	M5

3.1.1 Estratégia e Expressão de Busca

A estratégia de busca foi planejada de forma estruturada para assegurar abrangência e reprodutibilidade, conforme as diretrizes de Kitchenham ([Kitchenham and Charters, 2007](#)). O objetivo foi identificar estudos que abordassem técnicas de evasão com ênfase em *frameworks* de DBI, bem como ferramentas envolvidas na aplicação das técnicas.

As buscas foram realizadas em junho de 2025 e consideraram publicações no período de 2000 a 2025. Foram consultadas quatro bases digitais de ampla cobertura na área de Computação e Segurança da Informação: IEEE Xplore, Scopus, ACM Digital Library e SpringerLink. De forma complementar, o Google Scholar foi utilizado para localizar versões alternativas de artigos (preprints, cópias autorais) e literatura cinzenta relevante, como teses e dissertações.

A expressão de busca foi única, adaptada apenas em função da sintaxe de cada base, e organizada em quatro eixos: (i) instrumentação binária dinâmica, (ii) técnicas evasivas, (iii) ferramentas de empacotamento/proteção, (iv) transparência de *frameworks* DBI, e (v) análise dinâmica em sistemas Windows. A string utilizada foi:

```
("dynamic binary instrumentation" OR DBI OR "binary instrumentation") AND
("anti-instrumentation"OR "anti-analysis"OR "anti-disassembly"OR
"anti-reversing"OR "anti-tampering"OR "anti-debug*"OR "anti-VM"OR malware OR
evasion OR "environment detection"OR stealth OR transparency OR
"anti-evasion"OR "evasion mitigation"OR "evasion countermeasure"OR "evasion
detection"OR mitigation OR countermeasure OR bypass OR harden* OR resilient*
OR packer OR packing OR unpacking OR "software protector"OR "software
protection")
```

3.1.2 Critérios de Inclusão e Exclusão

A definição de critérios de inclusão (CI) e exclusão (CE) foi essencial para garantir consistência e relevância à RSL. Esses critérios foram aplicados de forma sistemática em todas as etapas de triagem (título, resumo, introdução, conclusão e leitura integral), conforme sintetizado na Tabela 3.4.

Tabela 3.4: Critérios de seleção dos artigos.

Critérios de Inclusão (CI)	Descrição
CI-1	Estudos que descrevam conceitos, técnicas ou ferramentas de instrumentação binária dinâmica (DBI).
CI-2	Publicações que abordem técnicas de evasão, incluindo <i>anti-instrumentation</i> , <i>anti-disassembly</i> , <i>anti-debug</i> ou <i>anti-VM</i> e as ferramentas empregadas na aplicação das técnicas, como protetores e empacotadores.
CI-3	Trabalhos que proponham ou avaliem contramedidas, em especial no contexto de <i>anti-DBI</i> e transparência de instrumentação.
CI-4	Estudos revisados por pares (conferências, periódicos e workshops), além de teses e dissertações com contribuição técnica relevante.
CI-5	Período de publicação compreendido entre 2000 e 2025.
Critérios de Exclusão (CE)	Descrição
CE-1	Trabalhos cujo texto completo não esteja disponível.
CE-2	Publicações em idiomas distintos de inglês, espanhol ou português.
CE-3	Artigos duplicados recuperados em mais de uma base de dados.
CE-4	Estudos cujo foco principal não esteja relacionado a evasão ou ambientes DBI.
CE-5	Relatos técnicos não revisados por pares, notas de aula, apresentações ou literatura cinzenta sem validação científica.

3.1.3 Bases de Dados e Resultados da Busca

As bases selecionadas foram IEEE Xplore, Scopus, ACM Digital Library e SpringerLink, por sua abrangência em conferências e periódicos de alto impacto nas áreas de Computação e Segurança da Informação, tais como IEEE S&P, RAID e ACSAC. O Google Scholar foi utilizado de forma complementar, sobretudo para identificar versões alternativas de artigos pagos ou não disponíveis.

A Tabela 3.5 resume o quantitativo de registros inicialmente identificados em cada base e as contagens referentes às etapas sequenciais do processo de triagem.

Tabela 3.5: Resultados das buscas por base e contagens por etapa de triagem.

Biblioteca Digital	Total identificados	Removidos na consolidação (duplicados)	Excluídos no Filtro 1	Excluídos no Filtro 2	Incluídos (final)
IEEE Xplore	311	0	297	4	10
Scopus	438	11	359	39	29
ACM DL	2440	5	2371	26	38
SpringerLink	486	12	433	25	16
TOTAL	3675	28	3460	94	93

A sequência de filtragem aplicada durante a revisão sistemática foi conduzida conforme os seguintes estágios:

- (i) **Identificação:** 3675 registros foram identificados nas quatro bibliotecas digitais selecionadas: IEEE Xplore, Scopus, ACM Digital Library e SpringerLink.
- (ii) **Consolidação e remoção de duplicatas:** 28 registros duplicados foram eliminados, totalizando 3647 registros únicos.
- (iii) **Filtro 1 (triagem por título e resumo):** 3460 registros foram excluídos por não atenderem aos critérios de escopo, resultando em 187 registros para avaliação preliminar.
- (iv) **Filtro 2 (leitura parcial):** 94 registros foram excluídos após leitura superficial da introdução e/ou conclusão, em razão de texto completo indisponível ou irrelevância temática, resultando em 93 estudos incluídos na análise final.

Os critérios de inclusão consideraram estudos que abordam diretamente técnicas de evasão, ferramentas e contramedidas aplicadas em contextos de instrumentação binária dinâmica, publicados em inglês, espanhol ou português, contendo resultados empíricos, metodologias reproduzíveis ou artefatos disponíveis publicamente. Os principais critérios de exclusão englobaram: escopo fora do tema, revisões sem contribuição empírica, comunicações curtas sem detalhamento metodológico, duplicatas e publicações com texto completo inacessível quando não foi possível obtê-lo por outros meios.

A Tabela 3.6 apresenta um resumo do processo de seleção segundo o modelo PRISMA.

3.2 Revisão da Literatura

Ferramentas baseadas em instrumentação binária dinâmica (DBI) têm sido amplamente empregadas em análise de código em tempo de execução. Contudo, à medida que se consolidaram como suporte a estudos de segurança e análise de software malicioso, também se tornaram alvo de técnicas de evasão cada vez mais sofisticadas. Nesta seção, discutimos os principais trabalhos da literatura, organizados em dois blocos: (i) técnicas de evasão em malware e empacotadores/protetores, e (ii) contramedidas e ferramentas de análise baseadas em DBI voltadas para malware evasivo. Cada bloco é apresentado de maneira cronológica, a fim de situar o leitor quanto ao avanço das técnicas evasivas e as ferramentas para neutralizá-las.

Tabela 3.6: Resumo do processo de seleção dos artigos (modelo PRISMA), com contagens sequenciais.

Etapa	Descrição e quantitativo
Identificação	3675 registros identificados nas bases de dados (IEEE Xplore, Scopus, ACM DL, Springer-Link).
Remoção de duplicatas	28 registros removidos, restando 3647 para triagem inicial.
Triagem (título e resumo)	3460 registros excluídos no primeiro filtro, restando 187 registros.
Avaliação de elegibilidade (leitura parcial)	94 registros excluídos no segundo filtro, restando 93 artigos selecionados para a análise final.
Inclusão	93 estudos incluídos na síntese qualitativa final.

3.2.1 Técnicas de Evasão em Malware e Empacotadores/Protetores

O trabalho de [Botacin et al. \(2017\)](#) apresenta uma visão abrangente do cenário de *malware* evasivo, propondo uma taxonomia de técnicas de anti-análise e introduzindo o conceito de *anti-anti-analysis*, isto é, mecanismos destinados a reconhecer e neutralizar tentativas de evasão antes da execução efetiva do código. A contribuição central está na análise estática em larga escala (mais de cem mil amostras), que permite observar diferenças de adoção de técnicas por contexto, mas que os próprios autores reconhecem como insuficiente para capturar o impacto real dessas técnicas em tempo de execução. O estudo mostra, contudo, que há um núcleo relativamente estável de verificações de ambiente (estrutura do sistema, efeitos colaterais de instruções, artefatos de análise) que tende a reaparecer em amostras modernas. A proposta desta dissertação se posiciona nesse mesmo espaço de problematização, mas deslocando o foco para o acompanhamento dinâmico dessas verificações, algo que o trabalho de [Botacin et al. \(2017\)](#) aponta como passo futuro.

[Park et al. \(2019\)](#) tratam de um subconjunto bem definido do problema: técnicas de *anti-debugging* em Windows e sua neutralização automática a partir de um ambiente de instrumentação binária dinâmica (Pin). Os autores catalogam 29 técnicas, mostram que parte delas ainda detecta o DBI e, a partir daí, propõem um ciclo “detectar–classificar–burlar” que lhes permite analisar 17 empacotadores comerciais de forma bem-sucedida. A relevância para o presente estudo está menos no conjunto exato de técnicas neutralizadas e mais na constatação de que o próprio DBI pode ser usado como plataforma para identificar e contornar evasões em tempo de execução, em vez de depender apenas de extensões de depuradores ou de heurísticas estáticas.

O levantamento de [Afianian et al. \(2018\)](#) é especialmente útil porque separa com clareza evasões dependentes de detecção (aquelas que primeiro tentam identificar VM, *sandbox*, depurador ou DBI) de evasões independentes de detecção (atrasos temporizados, exigência de interação humana, comportamentos acionados por contexto). Os autores mostram que o esforço dos *malwares* mais recentes está muito voltado à *fingerprinting* de ambientes automatizados e que respostas puramente reativas ou apenas “mais transparentes” tendem a ser ultrapassadas por novas variantes. Esse diagnóstico é compatível com a proposta desta dissertação: observar vários pontos do comportamento dinâmico (chamadas de API, memória, instruções) justamente para poder relacionar evidências de que o programa está sondando o ambiente de execução.

O estudo [Coccia et al. \(2022\)](#) demonstra, de forma empírica, que empacotadores comerciais frequentemente incorporam rotinas de *anti-debugging* e anti-VM, ainda que seu propósito declarado seja apenas compactação ou proteção de código: 12 de 20 empacotadores testados apresentaram ao menos uma técnica desse tipo. Também mostra que há um núcleo de verificações “clássicas” (por exemplo, consultas a APIs bem conhecidas), mas sem um padrão único que permita tratar todos os empacotadores da mesma forma. Esse resultado reforça a motivação geral deste trabalho: quando o executável está protegido, é precisamente o empacotador que passa a ser o vetor de evasão e, portanto, é sobre o comportamento dele, em tempo de execução, que a análise deve se concentrar.

[Galloro et al. \(2022\)](#) sistematizam 92 técnicas de evasão voltadas a ambientes instrumentados e as avaliam em 45,375 amostras de *malware* Windows. Como resultado do estudo, podemos destacar: (i) a proporção de amostras evasivas cresce pouco, mas cresce; e (ii) o perfil das técnicas muda ao longo do tempo, com algumas aparecendo quase exclusivamente em *malware* e não em software legítimo. O trabalho mostra também que ferramentas baseadas em DBI são adequadas para registrar o momento em que o binário tenta coletar artefatos do ambiente, exatamente o tipo de ponto de observação que este estudo explora.

[D’Elia et al. \(2022\)](#) abordam um ponto crítico para qualquer solução que queira usar DBI para análise de código malicioso: a transparência não pode ser presumida. Os autores reúnem e atualizam vários métodos de detecção de instrumentação e os aplicam aos principais frameworks (Pin, DynamoRIO, Dyninst, QBDI, QEMU em modo usuário), mostrando que todos deixam algum traço detectável — por tempo, por artefato de memória, por uso de APIs de depuração ou por defeitos de tradução. A conclusão deles é que, para análises de segurança, o ambiente de DBI pode normalizar respostas sensíveis por meio de contramedidas, ou é preciso registrar explicitamente essas sondagens como indícios de evasão. Essa conclusão se alinha com a linha desta dissertação.

[Muralidharan et al. \(2022\)](#) olham para o problema pela porta de entrada: o empacotamento. A partir de 24 mil PE analisados entre 2010 e 2021, os autores mostram que o empacotamento continua recorrente e é mais prevalente em amostras maliciosas do que em benignas; mostram também que há um crescimento de empacotadores menos tradicionais e de técnicas de *anti-unpacking* em tempo de execução. O ponto de contato com o presente trabalho está na consequência desse cenário: se o empacotamento passou a ser um mecanismo de evasão e não apenas de proteção de propriedade intelectual, então mecanismos de análise precisam observar o comportamento *depois* que o código é revelado em memória — o que, novamente, favorece abordagens dinâmicas baseadas em instrumentação.

[Xu et al. \(2023\)](#) avançam sobre o problema clássico de “o *malware* desviou o fluxo porque detectou o ambiente” propondo o Antitoxin, que combina contramedidas ativas para evasões conhecidas com exploração multi-caminho guiada por *taint* e probabilidade para alcançar ramos pouco prováveis (típicos de *logic bombs*). A contribuição deles é mostrar que é possível priorizar caminhos mais “suspeitos” a partir dos próprios pontos em que o binário consulta o ambiente. Essa ideia é compatível com o tipo de dados que este trabalho pretende coletar: uma vez identificados, em tempo de execução, os pontos de sondagem do ambiente, eles podem servir de origem para explorações direcionadas.

Por fim, [Park and Park \(2024\)](#) voltam ao tema da detecção de ambientes de DBI em Windows e mostram, com 24 técnicas implementadas, que há um repertório crescente de verificações que podem ser embutidas em código malicioso ou protegido para expor analisadores dinâmicos. O valor desse trabalho, para o contexto deste trabalho, está em tornar explícito que essas verificações são variadas (tempo, memória, APIs, inicialização) e que uma solução prática

precisa tanto conhecê-las quanto registrá-las quando ocorrem. É essa linha geral — detectar que *há* evasão, em vez de pressupor completa invisibilidade — que fundamenta a proposta que será apresentada nos capítulos seguintes.

3.2.2 Contramedidas e Ferramentas de Análise Baseadas em DBI Voltadas para Malware Evasivo

D'Alessio and Mariani (2015) apresentam o *PinDemonium*, um *unpacker* genérico para executáveis Windows baseado em instrumentação binária dinâmica com Intel Pin, cujo objetivo é automatizar a recuperação do código original a partir de binários empacotados ou ofuscados (D'Alessio and Mariani, 2015). A proposta parte de uma premissa observável em praticamente todos os esquemas de *runtime unpacking*: em algum momento da execução, o empacotador precisa escrever código em memória e, em seguida, transferir o fluxo de controle para essa região recém-gerada. O sistema monitora exatamente esse padrão “*write* → *execute*” e, quando ele ocorre, coleta *dumps* de memória e tenta reconstruir um PE funcional, incluindo a reconstituição da *Import Address Table* para casos em que o empacotador oculta ou posterga a resolução de APIs. Nos experimentos relatados, com 15 empacotadores conhecidos e 1,066 amostras coletadas do VirusTotal, o *PinDemonium* obteve 49% de sucesso pleno (binário reconstruído e executável) e 63% de sucesso útil (binário reconstruído e analisável, ainda que não idêntico ao original), o que é expressivo para uma abordagem não baseada em assinaturas. Os autores também registram os casos que falharam por motivos típicos de cenários reais — como evasão do ambiente instrumentado, detecção do próprio Pin ou técnicas de ofuscação de IAT fora do escopo — ressaltando que uma ferramenta de análise dinâmica precisa, além de detectar o momento do desempacotamento, lidar com tentativas do executável de reconhecer o ambiente de instrumentação. Essa constatação é relevante para o presente trabalho, que igualmente parte da ideia de observar o comportamento em tempo de execução para evidenciar etapas de proteção e evasão, preservando tanto quanto possível a transparência do ambiente de análise.

Rodríguez et al. (2016) investigam o problema do chamado *analysis-aware malware*, isto é, amostras capazes de reconhecer que estão sendo executadas em um ambiente isolado (sandbox, VM ou laboratório automatizado) e, a partir disso, alterar ou suprimir o comportamento malicioso. Os autores revisam inicialmente as principais técnicas de detecção de ambientes de análise — inspeção de artefatos de virtualização, consultas a recursos de depuração, medições temporais e verificação de interação humana — e, em seguida, propõem o *PinVMShield*, uma ferramenta baseada em DBI que intercepta essas sondagens e fornece respostas normalizadas, de modo que o binário analisado conclua estar em um ambiente não monitorado. A solução mantém a execução original do programa e permite injetar código em tempo de execução, o que a torna independente de linguagem e de compilador. Nos experimentos com amostras conhecidas de *malware* sensível a sandbox, o protótipo foi capaz de induzir a exposição do comportamento real, demonstrando que DBI pode ser usado não apenas para observar evasões, mas também para anulá-las. Os autores reconhecem, contudo, que se trata de um processo iterativo: à medida que ferramentas de análise adotam DBI para mascarar o ambiente, *malwares* tendem a incorporar também verificações específicas de DBI, o que reforça a necessidade de soluções capazes de atualizar continuamente o conjunto de contramedidas — linha de evolução com a qual o presente trabalho é compatível.

Polino et al. (2017a) partem da constatação de que, à medida que ferramentas de análise passaram a usar instrumentação binária dinâmica para obter rastros mais ricos, *malwares* também passaram a incorporar técnicas específicas de anti-instrumentação, distintas daquelas clássicas de anti-*debugging* e anti-*VM*. Os autores sistematizam as principais formas de expor um DBI em

Windows e as agrupam em quatro classes: (i) artefatos do *code cache*; (ii) artefatos de ambiente (módulos carregados, regiões injetadas, diferenças observáveis via `VirtualQuery` ou PEB); (iii) detecção do compilador JIT do próprio DBI; e (iv) medições de *overhead* baseadas em tempo. Sobre essa taxonomia, constroem um conjunto de contramedidas genéricas, implementadas no *Arancino* (sobre Intel Pin), que interceptam o ponto em que o binário tenta sondar o ambiente e devolvem valores coerentes com uma execução nativa. Com o mecanismo ativo, realizam a primeira medição em larga escala do fenômeno: em um conjunto de 7,006 amostras recentes, 15,6% ativaram pelo menos uma técnica de anti-instrumentação, evidenciando que esse tipo de evasão já estava disseminado naquele contexto temporal. Um resultado relevante é que o mesmo arcabouço de “ocultação do DBI” pôde ser reutilizado para construir um *unpacker* dinâmico mais robusto, capaz de lidar com empacotadores que verificam a presença de instrumentação durante o *unpacking*. Esse modelo — catalogar técnicas, inserir normalizações em tempo de execução e, a partir disso, medir o que o *malware* realmente faz — é diretamente aproveitável em estudos que também pretendem observar e mitigar comportamento evasivo em Windows apoiando-se no Intel Pin.

Santos Filho et al. (2017) abordam diretamente o avanço das técnicas de evasão voltadas a instrumentadores binários dinâmicos, mostrando que, à medida que o Pin e ferramentas derivadas se tornam solução de referência para análise, *malwares* passam a explorar artefatos específicos do próprio DBI para detectar ou contornar a instrumentação. Tomando o *PinVMShield* de Rodríguez et al. (2016) como base, os autores implementam contramedidas adicionais contra duas classes de evasão consideradas, na época, estado da arte: exploração do *code cache* e uso de *Thread Local Storage* (TLS) para sondar o ambiente instrumentado. A contribuição principal está em demonstrar que é possível mitigar essas detecções sem perdas significativas de desempenho, desde que o instrumentador intercepte e normalize consultas sensíveis do binário ao ambiente. Para trabalhos que, como o presente, pretendem analisar executáveis potencialmente protegidos e com comportamento *analysis-aware*, o estudo indica uma direção clara: a camada de instrumentação não deve apenas registrar o acesso a APIs e estruturas internas do sistema operacional, mas também devolver respostas coerentes com uma execução nativa, de modo a preservar a transparência da análise.

O trabalho de Martins (2016) aborda a identificação de *malware* metamórfico por meio de grafos de dependência, explorando a ideia de que transformações sintáticas no código podem alterar a aparência do binário sem modificar sua semântica essencial. A abordagem busca extrair características estruturais mais robustas contra variações metamórficas, permitindo reconhecer relações de similaridade entre amostras mesmo quando técnicas de reescrita ou reorganização de código são aplicadas. Embora o foco do estudo esteja na identificação estrutural de famílias de *malware*, sua contribuição é relevante para este trabalho por tratar de uma classe de evasão baseada na transformação do código, problema também presente em empacotadores e protetores comerciais.

Suk et al. (2018) investigam, de forma empírica, o funcionamento de um protetor comercial amplamente utilizado e propõem o *UnThemida*, uma ferramenta capaz de deofuscar programas totalmente protegidos por esse empacotador. O ponto de partida é a observação de que, mesmo quando múltiplas opções de ofuscação estão habilitadas, o protetor precisa em algum momento restaurar código e dados em memória para que o programa original execute; o *UnThemida* explora exatamente esse momento de *runtime unpacking*, usando instrumentação dinâmica para capturar o código recuperado e reconstruir o PE, incluindo técnicas para restaurar a *Import Address Table* mesmo quando ela foi ofuscada. Diferentemente de abordagens anteriores que lidavam apenas com um subconjunto de opções ou versões, o trabalho sistematiza padrões de fluxo de controle e de estruturas geradas pelo protetor para identificar quais opções foram

aplicadas e, assim, automatizar o processo. Para pesquisas que analisam protetores comerciais como vetores de evasão, como esta, esse estudo mostra que é viável tratar o empacotador não apenas como uma “caixa-preta”, mas como um software cujo comportamento pode ser observado e normalizado em tempo de execução.

No estudo de [D’Elia et al. \(2019\)](#), dedicado à segurança de DBI, os autores argumentam que algumas classes de evasão podem ser mitigadas forçando transferências de controle apenas para regiões previamente autorizadas, evitando execuções inviáveis decorrentes, por exemplo, de inconsistências na aplicação de políticas de NX ([D’Elia et al., 2019](#)). Entretanto, o verdadeiro desafio para motores de DBI surge quando o adversário consegue registrar tratadores de exceção e forçar acessos a áreas de memória que, do ponto de vista do sistema operacional, deveriam aparecer como não alocadas. Nesses cenários, a solução mais direta — inspecionar ou “escutar” todo acesso à memória — impõe um custo de execução elevado, que os DBIs atuais dificilmente absorvem sem perda significativa de desempenho. Os autores observam que avanços de plataforma, como memória apenas executável, podem tornar essa mitigação mais factível no futuro. Também apontam que a popularização da análise por DBI não implica necessariamente em uma ampliação imediata do número de evasões específicas para DBI: em muitos casos, desenvolvedores de *malware* continuam priorizando técnicas mais tradicionais de anti-sandbox, anti-VM e ofuscação, porque atingem uma base mais ampla de ferramentas de análise. Por fim, lembram que mesmo abordagens estruturalmente mais transparentes, como VMI ou SBI, também possuem superfícies de ataque próprias e precisam equilibrar transparência, desempenho e capacidade de instrumentação — isto é, não existe um modelo único de instrumentação totalmente imune à detecção, apenas escolhas de projeto com compromissos distintos.

Em trabalho posterior, [D’Elia et al. \(2020\)](#) propõem o *BluePill*, um ambiente de execução baseado em DBI voltado à dissecação manual de amostras altamente evasivas, conciliando dois requisitos que costumam ser tratados separadamente: ocultação e capacidade de intervenção fina ([D’Elia et al., 2020](#)). A solução oculta artefatos introduzidos pelo próprio processo de análise, intercepta sondagens típicas de evasão (tempo, virtualização, *debugger*, *hooks*) e ainda permite acrescentar análises auxiliares, como *taint analysis*, sem tornar o binário mais fácil de detectar. Por oferecer um ambiente extensível e ainda assim comparável, em termos de descrição, a *sandboxes* comerciais, o *BluePill* torna-se especialmente adequado para a etapa em que o analista passa a modificar o código em execução — cenário semelhante ao considerado neste trabalho, no qual se assume que o DBI também deixa artefatos e que estes precisam ser ocultados para viabilizar a observação do comportamento real.

[Lee et al. \(2021\)](#) enfrentam o problema prático de usar DBI em executáveis protegidos comercialmente: antes de analisar ou desempacotar, é necessário primeiro contornar as rotinas de anti-VM e anti-DBI embutidas pelo protetor. Os autores analisam cinco protetores amplamente usados e, a partir deles, extraem e classificam as verificações de ambiente (VM, *debugger*, artefatos de instrumentação) e as rotinas de finalização precoce, propondo para cada categoria um algoritmo de *bypass* baseado em instrumentação. O caráter empírico do estudo — que aplica os protetores sobre 1,573 executáveis e testa os *bypasses* — torna os resultados diretamente aplicáveis a sistemas de análise que pretendem operar em ambientes virtualizados e, ainda assim, observar o código protegido em execução. A conclusão central é que a análise dinâmica baseada em DBI, nesses cenários, só é viável se vier acompanhada de mecanismos sistemáticos de interceptação e normalização das sondagens de ambiente.

[Gozzini et al. \(2022\)](#) propõe o *PINvader*, um *framework* de análise dinâmica sobre Intel Pin direcionado à detecção e ao contorno de técnicas evasivas em binários Windows x64, com um aspecto singular: a avaliação é feita sobre um conjunto amplo de *goodware*, e não apenas sobre

malware. O sistema instrumenta em múltiplos níveis (APIs, instruções e exceções) e, a partir de uma taxonomia própria (técnicas temporais, de ambiente/sistema e de exploração de tratamento de exceções), executa o binário já com contramedidas ativas. Ao aplicar o *PINvader* em mais de 11 mil executáveis, o estudo demonstra elevada incidência de técnicas de *timing* (mais de 17,500 detecções), seguida de sondagens de ambiente e de usos de exceção, e mostra que classes legítimas de software também exibem comportamento evasivo. O resultado é importante porque evidencia que mecanismos de detecção e mitigação de evasão baseados em DBI podem — e devem — ser aplicados mesmo quando o alvo não é explicitamente malicioso.

Filho et al. (2022) apresentam um mapeamento abrangente do antagonismo entre *malwares* e *frameworks* de instrumentação binária dinâmica em ambiente Windows, identificando 26 técnicas de evasão especificamente direcionadas a DBI e mostrando que apenas 12 delas contam com contramedidas descritas na literatura. O estudo evidencia que o próprio DBI, ao inserir código previsível, manter páginas com permissões ampliadas (W+X) ou expor artefatos de execução, amplia a superfície de ataque e torna o ambiente de análise detectável; ao mesmo tempo, observa que as propostas de mitigação são, em geral, pontuais, pouco avaliadas em cenários realistas e raramente comparadas entre si quanto a custo e eficácia. Os autores apontam cinco lacunas principais: ausência de contramedidas realmente genéricas; escassez de experimentação com *malware* de fato evasivo; falta de medições sistemáticas de *overhead*; inexistência de estudos comparativos; e reduzido número de provas de conceito funcionais. Esse panorama converge com o objetivo desta dissertação, que busca fornecer evidências empíricas em cenários de análise reais, justamente no ponto indicado pelos autores como mais frágil.

Crăciun et al. (2023) partem do diagnóstico de que DBIs consolidados (como Pin e DynamoRIO) oferecem o nível de controle necessário, mas não são, por padrão, suficientemente transparentes nem triviais de operar em contextos de *malware* evasivo. Para isso, propõem o *COBAI*, um *framework* de instrumentação concebido com dois requisitos de base: transparência (isto é, esconder a presença do DBI) e operacionalidade (orquestrar múltiplos módulos e ferramentas sem exigir reimplementação frequente). O diferencial está na combinação desse *framework* com uma suíte de testes de evasão que agrega cenários públicos (como *pafish* e *al-khaser*) e cenários próprios, permitindo avaliar o grau de transparência obtido. Nos experimentos, o *COBAI* mitigou aproximadamente 96% dos ataques de transparência ao DBI e 100% dos de ambiente, indicando que uma abordagem em camadas (DBI → sistema operacional → sandbox) reduz significativamente o espaço de detecção.

O *Pinicorn*, de Lee et al. (2024), foca em um problema que muitos *unpackers* genéricos ainda não resolvem bem: empacotadores que, além de removerem a IAT e deslocarem o ponto de entrada (OEP), inserem *trampoline code* tanto na entrada do programa quanto nas posições de APIs, por vezes de forma sensível a argumentos. Nesses casos, o *dump* de memória obtido pela análise dinâmica não é suficiente para reconstruir automaticamente nem o OEP real nem a tabela de importação, porque o que está em memória são *stubs* intermediários e não as APIs propriamente ditas. O *Pinicorn* resolve o problema em duas fases: (i) executa o binário sob DBI para atravessar as rotinas de anti-análise e capturar o programa restaurado em memória; e (ii) reexecuta, emulando, todos os *trampolins* identificados, de forma a determinar quais APIs estavam ocultas e qual OEP havia sido camuflado. Com isso, consegue desempacotar amostras protegidas por empacotadores recentes e demonstra que o desafio não é apenas “desempacotar”, mas remover simultaneamente a obfuscação aplicada ao OEP e à IAT.

Por fim, Zhang et al. (2025) apresentam o *Devmp*, que trata especificamente da ofuscação por virtualização de código, típica de protetores comerciais como *VMProtect* e *Code Virtualizer*. Nesses cenários, o binário executa *bytecodes* próprios, interpretados por um *handler loop*, o que

torna irrelevante a simples coleta de instruções x86/x64. O *Devmp* combina DBI, para obter traços reais de execução, com execução simbólica *offline*, para particionar e analisar os *handlers* da máquina virtual, resolver saltos virtuais e extrair instruções virtuais com semântica completa. Em avaliações com duas versões do *VMProtect*, o protótipo superou ferramentas anteriores (como *VMP Analysis Plugin* e *No Vmpy*) tanto em taxa de reconhecimento de instruções virtuais quanto em correção semântica, indicando que estratégias baseadas em DBI podem ser estendidas de forma eficaz ao domínio de virtualização de código, desde que acompanhadas de uma fase analítica posterior capaz de reconstruir o nível lógico que o protetor tentou ocultar.

3.3 Discussão

Neste capítulo, apresentamos uma revisão abrangente da literatura relacionada à análise de *malwares*, com foco especial na Instrumentação Binária Dinâmica (DBI) e nas técnicas evasivas empregadas por softwares maliciosos modernos. Os estudos analisados evidenciam a crescente sofisticação dos *malwares* em utilizar métodos avançados de *anti-debugging*, *anti-VM* e *anti-instrumentação* para evitar detecção e análise.

Os trabalhos revisados mostram um ponto comum: quando o DBI passa a ser usado de forma sistemática para análise de *malware* em Windows, ele próprio se torna alvo de detecção e evasão. Isso aparece em contextos distintos — desempacotamento dinâmico [D'Alessio and Mariani \(2015\)](#); [Suk et al. \(2018\)](#); [Lee et al. \(2024\)](#), mascaramento de ambientes de análise [Rodríguez et al. \(2016\)](#); [Santos Filho et al. \(2017\)](#), endurecimento do próprio DBI [Polino et al. \(2017a\)](#); [Filho et al. \(2022\)](#) e ambientes de dissecação mais interativos [D'Elia et al. \(2020\)](#); [Crăciun et al. \(2023\)](#) — mas a mensagem é a mesma: não basta “instrumentar e observar”; é preciso instrumentar, observar e responder de forma plausível às sondagens feitas pelo binário. Essa conclusão é diretamente relevante para esta dissertação, que também depende de registrar chamadas sensíveis (*VirtualQuery*, leitura de PEB, medições de tempo) sem que essas mesmas chamadas terminem a execução ou ocultem o comportamento malicioso.

Outro ponto recorrente é que a literatura já mapeou bem *quais* técnicas de evasão existem (26 em [Filho et al. \(2022\)](#)), quatro grandes classes em [Polino et al. \(2017a\)](#)), mas há menos resultados sobre *como* essas técnicas se manifestam em amostras reais x64 protegidas por empacotadores comerciais. Vários autores notam a carência de avaliações em cenários realistas, com VM e com protetores que combinam anti-VM e anti-DBI ([Lee et al., 2021](#)), exatamente o tipo de cenário que esta dissertação pretende documentar de forma empírica. Em termos de contribuição, este trabalho se apoia nesse vácuo: partimos de um catálogo já consolidado, mas aplicamos o modelo em um ambiente próximo ao de uso real.

Há também uma lição metodológica dos trabalhos mais recentes: transparência total de DBI é, na prática, um objetivo assintótico. Soluções como *Arancino*, *PINvader* e *COBAI* mostram que é possível reduzir muito os artefatos, mas sempre ao custo de mais interceptações e, portanto, mais *overhead*. Isso reforça a escolha desta dissertação por uma abordagem orientada a *observabilidade e manipulação*: registrar *quando* a evasão é acionada, para identificar indícios e, eventualmente, aplicar contramedidas.

Por fim, vários estudos ligados a desempacotamento e deofuscação dinâmica [Suk et al. \(2018\)](#); [Lee et al. \(2024\)](#); [Zhang et al. \(2025\)](#) apontam que o DBI é especialmente valioso quando funciona como ponto de coleta rico — não apenas para “rodar o binário”, mas para capturar código restaurado, reconstruir IAT/OEP e até extrair instruções virtuais. Essa visão é coerente com o propósito desta dissertação: usar o DBI como camada de observação que torna explícitas as etapas de proteção e evasão, e sobre essa base aplicar, quando necessário, normalizações e contramedidas incrementais.

Em síntese, os estudos analisados indicam que as técnicas evasivas mais recorrentes em *malwares* e protetores comerciais concentram-se em cinco grupos principais: (i) técnicas de anti-depuração, baseadas em APIs e estruturas internas do processo, como `IsDebuggerPresent`, `CheckRemoteDebuggerPresent`, `NtQueryInformationProcess` e `PEB` [Park et al. \(2019\)](#); [Lee et al. \(2021\)](#); (ii) técnicas anti-VM e anti-*sandbox*, voltadas à identificação de artefatos do ambiente de análise, como chaves de Registro, módulos, dispositivos, drivers e características de virtualização [Afianian et al. \(2018\)](#); [Galloro et al. \(2022\)](#); [Park and Park \(2024\)](#); (iii) medições temporais e estratégias de *stalling*, utilizadas para detectar *overhead* ou atrasar a execução em ambientes monitorados [Afianian et al. \(2018\)](#); [Gozzini et al. \(2022\)](#); (iv) técnicas de anti-instrumentação, que exploram artefatos introduzidos por *frameworks* DBI, como *code cache*, JIT, permissões de memória e inconsistências de tradução [Polino et al. \(2017a\)](#); [Filho et al. \(2022\)](#); [D'Elia et al. \(2022\)](#); [Park and Park \(2024\)](#); e (v) mecanismos de empacotamento, ofuscação, anti-*unpacking* e virtualização de código, que ocultam o comportamento real até o tempo de execução [Muralidharan et al. \(2022\)](#); [D'Alessio and Mariani \(2015\)](#); [Suk et al. \(2018\)](#); [Lee et al. \(2024\)](#); [Zhang et al. \(2025\)](#). Essa síntese reforça que a análise de *malware* evasivo exige uma abordagem dinâmica, capaz de observar chamadas de API, acessos à memória, fluxo de instruções e efeitos temporais, ao mesmo tempo em que considera a necessidade de contramedidas para preservar a transparência do ambiente instrumentado.

Capítulo 4

Técnicas Evasivas

Neste capítulo, examinamos algumas das técnicas evasivas utilizadas por *malwares* para evitar a detecção e análise por parte de profissionais de segurança cibernética. Essas técnicas têm evoluído significativamente, tornando os *malwares* cada vez mais sofisticados e capazes de ocultar seu comportamento malicioso. Ao evadir sistemas de segurança e dificultar a análise, os *malwares* aumentam sua longevidade e o potencial de causar danos.

Compreender essas técnicas é fundamental para o desenvolvimento de contramedidas eficazes e para a melhoria das práticas de análise de segurança. Ao explorar os métodos que os *malwares* empregam para se esconder e frustrar a detecção, podemos aprimorar as estratégias de defesa e fortalecer a resiliência dos sistemas contra ameaças cibernéticas emergentes.

4.1 Anti-Disassembly

Interpretar o código “desmontado” (*disassembled*) é um grande desafio, principalmente porque ele carece de informações contextuais, como a identificação de variáveis e as delimitações claras das funções ou rotinas de código.

Os trabalhos de Branco et al. (2012), Singh and Singh (2018) e Cheng et al. (2021) destacam diferentes técnicas *anti-disassembly*. Essas táticas atrasam ou até impedem a análise completa do código, aumentando o nível de habilidade necessário para que os analistas compreendam o comportamento do *malware*. Portanto, a eficácia dessas técnicas não reside apenas na confusão gerada, mas também no tempo extra exigido dos analistas, muitas vezes prejudicando investigações sensíveis ao tempo, como em ataques críticos de *malwares*.

Branco et al. (2012) destacam que a desmontagem de binários pode ser feita por meio de abordagens estáticas e dinâmicas. Na desmontagem estática, existem duas técnicas principais: a varredura linear e a travessia recursiva. A varredura linear analisa o código *byte a byte*, mas pode gerar ruído ao interpretar dados como se fossem código. Em contrapartida, a travessia recursiva segue o fluxo de controle do programa, mas enfrenta dificuldades em prever corretamente todas as ramificações de execução, deixando partes do código não desmontadas. Embora cada abordagem de desmontagem tenha suas vantagens, as técnicas *anti-disassembly* podem explorar essas limitações, gerando códigos que burlam os algoritmos de desmontagem, principalmente ao inserir dados entre as instruções de código. Dessa forma, quaisquer dados incorporados no código são interpretados como uma instrução, resultando em um código desmontado final incorreto (Botacin et al., 2017).

No entanto, tais técnicas são projetadas para confundir o desmontador, gerando código incorreto e prolongando significativamente o processo de análise. Embora as abordagens de *anti-disassembly* não sejam invencíveis, elas acarretam um nível elevado de complexidade e aumentam o tempo necessário para que os analistas conduzam a investigação de forma completa.

4.1.1 Instruções de Salto com o Mesmo Alvo

Sikorski and Honig (2012) explicam que o uso de instruções de salto com o mesmo alvo envolve o uso de duas instruções de salto condicional consecutivas que apontam para o mesmo destino. Por exemplo, se uma instrução `jz loc_512` for seguida por uma instrução `jnz loc_512`, o programa sempre saltará para `loc_512`, independentemente da condição. Na prática, essa combinação de `jz` com `jnz` funciona como um salto incondicional, mas o desassemblador não o reconhece dessa forma, pois desmonta uma instrução por vez. Assim, quando o desassemblador encontra a instrução `jmp`, ele continua desmontando o ramo “falso” dessa instrução, mesmo que nunca seja executado.

74 03	jz	short near ptr loc_4011C4+1	
75 01	jnz	short near ptr loc_4011C4+1	
	loc_4011C4:		; CODE XREF: sub_4011C0
			; 2sub_4011C0+2j
E8 58 C3 90 90	call	near ptr 90D0D521h	

Figura 4.1: Exemplo do uso de instruções de salto com o mesmo alvo. (Sikorski and Honig, 2012)

No exemplo apresentado na Figura 4.1, a instrução logo após os dois saltos condicionais parece ser uma instrução `call` (1), começando com o *byte* `0xE8`. No entanto, ambos os saltos condicionais apontam para 1 *byte* após o *byte* `0xE8`.

A Figura 4.2 mostra graficamente a sequência de *bytes* neste exemplo.

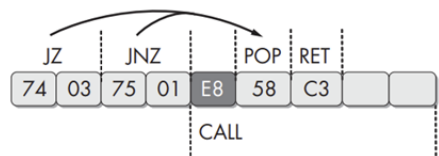


Figura 4.2: Instrução `jz` seguida por uma instrução `jmp`. (Sikorski and Honig, 2012)

O uso de instruções de salto com o mesmo alvo confunde o desmontador e dificulta a análise, pois as ferramentas de desmontagem acabam gerando interpretações incorretas do fluxo de controle do código.

4.1.2 Instruções com Condição Constante

Sikorski and Honig (2012) apresentam uma técnica chamada de instruções com condição constante, que consiste em usar uma única instrução de salto condicional onde a condição será sempre a mesma (Figura 4.3).

33 C0	xor	eax, eax
74 01	jz	short near ptr loc_4011C4+1
loc_4011C4:		; CODE XREF: 004011C2j
		; DATA XREF: .rdata:004020ACo
E9 58 C3 68 94	jmp	near ptr 94A8D521h

Figura 4.3: Exemplo do uso de instruções com condição constante. (Sikorski and Honig, 2012)

O código começa com a instrução `xor eax, eax`, que zera o registrador EAX e, como efeito colateral, define o *zero flag*. A instrução seguinte é um salto condicional (`jz`), que ocorre se o *zero flag* estiver ativado. Neste caso, o salto sempre ocorrerá, pois o *zero flag* foi garantidamente ativado pela instrução anterior (Figura 4.4).

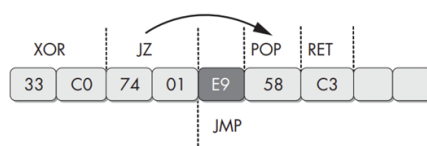


Figura 4.4: Falso condicional de `xor` seguido por uma instrução `jz`. (Sikorski and Honig, 2012)

Essa técnica é eficaz para enganar o desmontador, levando-o a interpretar erroneamente o fluxo de controle do código. Embora pareça ser um salto condicional, na verdade, ele funciona como um salto incondicional, uma vez que a condição sempre será verdadeira. O desmontador, entretanto, processará o “ramo falso” da instrução primeiro, o que resultará em código conflitante com o “ramo verdadeiro”. Como o desmontador confia mais no primeiro ramo que processa, essa técnica de evasão gera confusão e faz com que o código desmontado não reflita a realidade da execução do programa.

4.1.3 Mudança de Fluxo de Controle do Programa (Program Control Flow Change)

Branco et al. (2012) descrevem uma técnica de evasão chamada Mudança de Fluxo de Controle do Programa, que envolve a mudança incondicional do fluxo de controle de um programa para evitar que determinadas áreas do código sejam alcançadas em tempo de execução. Essas áreas podem conter outras técnicas de *anti-disassembly*, tornando-as inacessíveis durante a execução do programa, mas desassembladores que utilizam a abordagem de varredura linear ainda desmontam essas áreas, resultando em código incorreto.

Um exemplo dessa técnica, apresentado na Listagem 4.1, é o uso de um salto incondicional (`JMP`) para pular uma área que contém *bytes* de lixo (*garbage bytes*), que são usados para enganar o desassemblador.

Listagem 4.1: Exemplo de código para a técnica mudança de fluxo de controle do programa

```

1  jmp .destination
2  db 0x6a ; garbage byte technique
3 .destination:
4  ; rest of the code
5  pop eax
6  ...

```

Nesse caso, a instrução `jmp .destination` faz o salto para um local válido, mas a área intermediária, que contém o `byte` de lixo `0x6a`, será desmontada pelo desassemblador, resultando em uma saída incorreta, conforme a Listagem 4.2.

Listagem 4.2: Exemplo de código desmontado para a técnica mudança de fluxo de controle do programa

```
1 eb 01 jmp 0x401003
2 6a 58 push 0x58
```

O código desmontado mostra que o desassemblador interpreta o `byte` de lixo como uma instrução `push 0x58`, o que compromete a precisão do código desmontado. Essa técnica explora as suposições feitas pelos desassembladores, especialmente os que utilizam varredura linear, e compromete a saída resultante, gerando uma desmontagem incorreta que pode confundir os analistas.

4.1.4 Ofuscação de APIs

Cheng et al. (2021) mencionam a ofuscação de *APIs* como uma estratégia comum de *anti-disassembly*. Essa técnica remove os metadados das *APIs* importadas do cabeçalho *PE* do *malware*, complicando a identificação das chamadas de função. Mesmo quando o código é desmontado, o desassemblador falha em identificar corretamente as *APIs*, tornando a análise de comportamento mais difícil. Neste sentido, a ofuscação de *APIs* é uma das técnicas mais eficazes para esconder o comportamento malicioso e evitar que analistas identifiquem rapidamente as ações do *malware*.

Na Figura 4.5, Cheng et al. (2021) explicam que o “00000000” na visualização do disco representa um espaço reservado; o Carregador do Windows irá substituí-lo por um endereço de *API* (por exemplo, “7C805D86”) no momento do carregamento do programa de acordo com a interrupção *INT*.

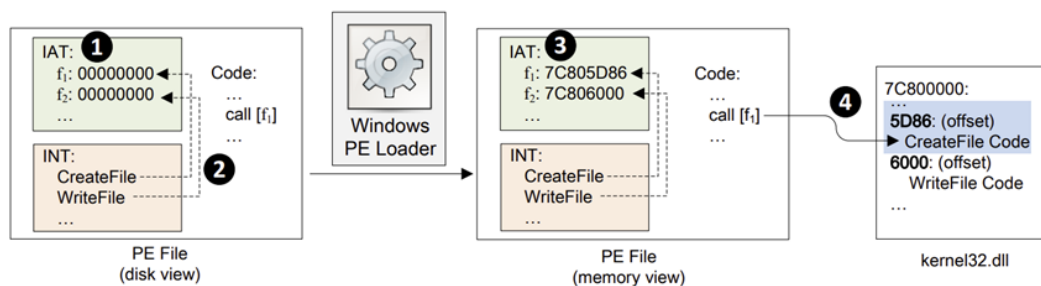


Figura 4.5: Exemplo de resolução padrão da *API*. (Cheng et al., 2021)

4.1.5 Desafios para a Detecção Usando o Instrumentador Pin

O *Pin* é uma ferramenta de instrumentação dinâmica, o que significa que ele examina o código em tempo de execução, monitorando as instruções conforme elas são executadas. No entanto, técnicas de *anti-disassembly* geralmente operam para enganar desmontadores estáticos, tentando confundir o processo de decodificação do código binário sem que ele seja realmente executado. Portanto, o *Pin* não pode ser utilizado diretamente para a análise estática do código *assembly*, já que ele não trabalha com o código binário fora do contexto da execução. Esta limitação do *Pin* pode ser um desafio porque, para detectar essas técnicas, é necessário analisar como o código se

comporta estaticamente, enquanto o *Pin* só tem acesso ao código durante a execução. Algumas técnicas de *anti-disassembly* podem simplesmente nunca ser ativadas durante a execução normal.

4.2 Anti-Debug

As técnicas de *anti-debugging* têm sido amplamente utilizadas para impedir ou dificultar a análise de softwares por meio de depuradores, desempenhando um papel muito importante no campo da engenharia reversa. A principal função dessas técnicas é detectar a presença de depuradores e modificar o comportamento do programa, retardando ou interrompendo o processo de análise (Park et al., 2019; Zhang, 2021; Polino et al., 2017a).

Zhang (2021) oferece uma visão sobre a divisão das técnicas de *anti-debugging* em duas categorias: estática e dinâmica. O *anti-debugging* estático verifica o estado do sistema antes da execução do programa, buscando por sinais da presença de depuradores. Zhang destaca que, apesar de ser de fácil implementação, essas técnicas são mais suscetíveis à detecção e *bypass*. Por outro lado, o *anti-debugging* dinâmico foca na observação do comportamento do programa em tempo de execução, utilizando inferências indiretas para identificar anomalias que indicam a presença de ferramentas de depuração. Embora seja mais difícil de contornar, Zhang aponta que o *anti-debugging* dinâmico é mais complexo de implementar.

A evolução dessas técnicas está diretamente relacionada ao aumento da sofisticação dos *malwares* modernos. Quando um *malware* detecta a presença de um depurador ou *sandbox*, ele pode alterar seu comportamento para frustrar os analistas e dificultar a compreensão completa do código malicioso (Polino et al., 2017a). Os autores ressaltam que existem centenas de técnicas de *anti-debugging* e que o número de técnicas continua a crescer.

Por outro lado, Park et al. (2019) destacam que as técnicas de *anti-debugging* geralmente estão presentes em *malwares* protegidos por programas comerciais como *Themida* e *VMProtect*, que utilizam também criptografia e ofuscação de código para proteger o *malware* contra análise. Assim, essas técnicas complicam significativamente a análise ao evitar a depuração direta do código.

Embora estudos como Park et al. (2019); Zhang (2021); Polino et al. (2017a); Yason (2007); Ferrie (2011) mostrem que existam métodos eficazes para contornar as técnicas de *anti-debugging*, essa é uma área que exige constante atualização e inovação. Por este motivo, é importante desenvolver as habilidades dos analistas para enfrentar novas técnicas de *anti-debugging* que podem surgir no futuro. Essa área está em constante evolução, e é importante que os analistas de *malware* se mantenham atualizados e preparados para lidar com essas ameaças em constante mudança.

4.2.1 PEB.BeingDebugged Flag (Função IsDebuggerPresent)

A técnica *PEB.BeingDebugged Flag* (Zhang, 2021; Ferrie, 2011; Yason, 2007) é uma das formas mais básicas de detecção de depuradores em sistemas *Windows*, amplamente usada por *malwares* para identificar se estão sendo executados em um ambiente de depuração. Essa técnica envolve a verificação de uma *flag* específica, o *BeingDebugged*, que está presente no *Process Environment Block* (*PEB*), uma estrutura de dados que o *Windows* mantém para cada processo em execução. O *flag BeingDebugged* indica se o processo atual está sendo depurado por um depurador em modo usuário.

A API *IsDebuggerPresent()* da biblioteca *kernel32.dll* verifica esse *flag* no *PEB* para determinar se o processo está sob depuração. O código de implementação da API acessa o *Thread Environment Block* (*TEB*), que contém o endereço do *PEB*, e, em seguida, acessa o campo *Be-*

ingDebugged localizado no deslocamento 0x02 dentro do *PEB*. O código em *assembly* mostrado na Listagem 4.3 segue esses passos:

Listagem 4.3: Implementação da API do Windows que acessa o TEB (Yason, 2007)

```
1 mov  eax, large fs:18h      ; Acessa o TEB
2 mov  eax, [eax+30h]        ; Acessa a PEB dentro do TEB
3 movzx eax, byte ptr [eax+2] ; Verifica flag BeingDebugged (offset 0x02 no PEB)
4 retn                      ; Retorna o valor de BeingDebugged
```

No código apresentado na Listagem demonstra que, se o valor retornado for diferente de zero, significa que o processo está sendo depurado.

Além do uso direto da *API IsDebuggerPresent()*, muitos *malwares* e empacotadores de código (*packers*) optam por acessar o *flag BeingDebugged* diretamente no *PEB*, em vez de chamar a *API*. Isso ocorre porque analistas de *malware*, durante a engenharia reversa, podem colocar *breakpoints* ou modificar a *API IsDebuggerPresent()* para evitar a detecção. A verificação direta do *flag* no *PEB* evita esse tipo de interferência.

Exemplo de Detecção: O código da Listagem 4.4 ilustra duas maneiras de identificar se um depurador está presente: usando a chamada para a *API IsDebuggerPresent()* e acessando diretamente o *flag BeingDebugged* no *PEB*.

Listagem 4.4: Verificação com a API IsDebuggerPresent (Yason, 2007)

```
1 call  [IsDebuggerPresent]  ; Chama a API IsDebuggerPresent()
2 test  eax, eax              ; Verifica o valor retornado
3 jnz   .debugger_found      ; Se diferente de zero, o depurador foi encontrado
```

Listagem 4.5: Verificação direta no PEB (Yason, 2007)

```
1 mov  eax, dword [fs:0x30]  ; EAX recebe o endereço do TEB e acessa o PEB
2 movzx eax, byte [eax+0x02] ; Verifica a flag BeingDebugged diretamente
3 test  eax, eax              ; Verifica o valor de BeingDebugged
4 jnz   .debugger_found      ; Se diferente de zero, o depurador foi encontrado
```

Na Listagem 4.5, se o valor do *BeingDebugged* for 1, indica que o processo está sendo depurado, e o código pode, então, alterar seu comportamento para evitar a análise, seja modificando a execução do programa ou encerrando-o.

4.2.2 PEB.NtGlobalFlag

A técnica *NtGlobalFlag* (Ferrie, 2011; Yason, 2007; Zhang, 2021) é usada para detectar depuradores verificando o campo *NtGlobalFlag* dentro do *Process Environment Block (PEB)*. Esse campo contém uma série de *flags* que indicam se o processo está sendo depurado. Quando um depurador cria um processo, três *flags* específicas são definidas: *FLG_HEAP_ENABLE_TAIL_CHECK* (0x10), *FLG_HEAP_ENABLE_FREE_CHECK* (0x20) e *FLG_HEAP_VALIDATE_PARAMETERS* (0x40). Ao verificar se essas *flags* estão definidas, o *malware* pode determinar se está sendo depurado.

O *NtGlobalFlag* está localizado no deslocamento 0x68 no *PEB* em sistemas *Windows* de 32 bits e no deslocamento 0xBC em sistemas de 64 bits. Abaixo, nas Listagens 4.6 e 4.7, estão os exemplos básicos de código *assembly* que verificam essas *flags* em arquiteturas de 32 e 64 bits:

Listagem 4.6: Verificação da NtGlobalFlag em sistemas de 32 bits (Ferrie, 2011)

```
1 mov  eax, fs:[30h]         ; Acessa o PEB
2 mov  al, [eax+68h]         ; Verifica o campo NtGlobalFlag
3 and  al, 70h               ; Filtra as flags relacionadas a depuracao
4 cmp  al, 70h               ; Compara com o valor esperado
5 je   being_debugged        ; Se flags encontradas, o processo esta sendo depurado
```

Listagem 4.7: Verificação da NtGlobalFlag em sistemas de 64 bits (Ferrie, 2011)

```

1 push 60h
2 pop rsi
3 gs:lodsq          ; Acessa o PEB em um ambiente de 64 bits
4 mov al, [rsi*2+rax-14h] ; Verifica o campo NtGlobalFlag
5 and al, 70h
6 cmp al, 70h
7 je being_debugged

```

Essa técnica verifica se as *flags* específicas de depuração estão ativas, o que indica a presença de um depurador. Embora os depuradores possam tentar mascarar essas *flags* ao redefini-las para zero, há várias maneiras de alterá-las, incluindo o uso de valores no registro do *Windows*. Por exemplo, as chaves de registro em HKLM\System\CurrentControlSet\Control\Session Manager OU HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options podem definir o valor de *NtGlobalFlag* para processos específicos ou para todos os processos do sistema.

Uma maneira de contornar essa técnica seria o depurador alterar o valor de *NtGlobalFlag* antes de retomar a execução do processo. No entanto, o *malware* pode usar truques, como verificar o valor no registro e compará-lo com o valor no campo *NtGlobalFlag* para detectar inconsistências e identificar a presença de um depurador.

4.2.3 FindWindow

A técnica *FindWindow* (Yason, 2007; Sikorski and Honig, 2012) é usada por *malwares* para identificar a presença de depuradores no sistema, por meio de janelas criadas por esses aplicativos. A função *FindWindow()*, da biblioteca *user32.dll*, é frequentemente utilizada para esse propósito, pois permite que o *malware* busque por janelas com nomes de classe específicos que correspondem a depuradores conhecidos, como *OllyDbg* ou *WinDbg*. A Listagem 4.8 apresenta um exemplo do uso da função *FindWindow()* na detecção de depuradores.

Listagem 4.8: Verificação FindWindow para OllyDbg em C (Sikorski and Honig, 2012)

```

1 if (FindWindow("OLLYDBG", 0) == NULL)
2 {
3     //Debugger Not Found
4 }
5 else
6 {
7     //Debugger Detected
8 }

```

Abaixo, nas Listagens 4.9 e 4.10 estão os códigos *assembly* que correspondem ao exemplo da Listagem 4.8:

Listagem 4.9: Verificação FindWindow para OllyDbg (Yason, 2007)

```

1 push NULL          ; Parametro para o titulo da janela (nao necessario aqui)
2 push .szWindowClassOllyDbg ; Classe da janela "OLLYDBG"
3 call [FindWindowA]    ; Chama a funcao FindWindow para buscar a janela
4 test eax, eax        ; Verifica se a janela foi encontrada (eax = 0 se nao
                        encontrada)
5 jnz .debugger_found  ; Se eax for diferente de zero, um depurador foi encontrado

```

Listagem 4.10: Verificação FindWindow para WinDbg (Yason, 2007)

```

1 push NULL          ; Parametro para o titulo da janela
2 push .szWindowClassWinDbg ; Classe da janela "WinDbgFrameClass"
3 call [FindWindowA]    ; Chama FindWindow para buscar a janela do WinDbg
4 test eax, eax        ; Verifica se a janela foi encontrada
5 jnz .debugger_found  ; Se eax for diferente de zero, um depurador foi encontrado

```

O código nos exemplos verifica se essas janelas estão presentes no sistema. Se o *FindWindow()* encontrar uma janela com a classe *OLLYDBG* para o depurador *OllyDbg* (Listagem 4.9) ou *WinDbgFrameClass* para o depurador *WinDbg* (Listagem 4.10), ele detecta que um depurador está em execução e, então, pode alterar o comportamento do *malware*, interromper sua execução ou implementar outras medidas evasivas. Caso contrário, se a função não encontrar essas janelas, o *malware* continua sua execução normalmente.

4.2.4 Técnicas de Medição de Tempo

As *Técnicas de Medição de Tempo* (Ferrie, 2011; Zhang, 2021) são estratégias utilizadas por *malwares* para detectar se estão sendo executados sob um depurador, aproveitando o fato de que a execução sob depuração geralmente é mais lenta devido ao *overhead* associado ao tratamento de eventos pelo depurador e à execução passo a passo das instruções. Essa técnica mede o tempo gasto entre a execução de instruções ou chamadas de funções para determinar se há um atraso significativo, que pode indicar a presença de um depurador.

Quando um processo é depurado, o tempo gasto para a execução de instruções é maior devido à intervenção do depurador, que pode pausar o programa ou inserir pausas para análise. O *malware* pode utilizar várias funções e instruções para medir o tempo de execução e comparar com os tempos esperados durante uma execução normal.

4.2.4.1 Função GetTickCount

A função *GetTickCount()* da API do Windows retorna o número de milissegundos desde a última reinicialização do sistema. Podemos observar no exemplo da Listagem 4.11 como o *malware* pode medir o tempo de execução entre duas chamadas a essa função para detectar atrasos causados pela depuração (Ferrie, 2011).

Listagem 4.11: Função *GetTickCount()* da API do Windows (Ferrie, 2011)

```
1 call GetTickCount      ; Chama a funcao GetTickCount
2 xchg ebx, eax          ; Troca o valor retornado com EBX
3 call GetTickCount      ; Chama GetTickCount novamente
4 sub eax, ebx           ; Calcula a diferenca
5 cmp eax, 10h           ; Compara com um limite (10h)
6 jnbe being_debugged    ; Se a diferenca for maior, assume-se que o depurador esta presente
```

4.2.4.2 Função QueryPerformanceCounter

A função *QueryPerformanceCounter()* fornece uma medição mais precisa do tempo de execução, usando o contador de desempenho de hardware. A Listagem 4.12 mostra como o *malware* pode usar essa função para medir o intervalo de tempo com maior precisão (Ferrie, 2011).

Listagem 4.12: Função *QueryPerformanceCounter()* da API do Windows (Ferrie, 2011)

```
1 mov esi, offset l1
2 push esi
3 call QueryPerformanceCounter
4 mov edi, offset l2
5 push edi
6 call QueryPerformanceCounter
7 mov eax, [edi]
8 sub eax, [esi]
9 mov edx, [edi+4]
10 sbb edx, [esi+4]
11 cmp eax, 10h
12 jnbe being_debugged
```

4.2.4.3 Instrução RDTSC

A instrução *RDTSC* (*Read Time-Stamp Counter*) lê o contador de marcação de tempo do processador, fornecendo uma medição de tempo com alta precisão. É uma abordagem direta para medir o tempo entre instruções (Ferrie, 2011). A Listagem 4.13 apresenta um exemplo do uso da instrução *RDTSC* para a detecção de depuradores.

Listagem 4.13: Uso da instrução RDTSC para detecção de depurador (Ferrie, 2011)

```
1 rdtsc          ; Le o contador de marcação de tempo
2 xchg esi, eax  ; Troca o valor com ESI
3 rdtsc          ; Le o contador novamente
4 sub eax, esi    ; Calcula a diferença
5 cmp eax, 500h   ; Compara com um limite
6 jnbe being_debugged ; Se a diferença for maior, assume-se que o depurador está presente
```

4.2.4.4 Outras Funções de Tempo

Funções adicionais como `GetLocalTime()`, `GetSystemTime()`, e `timeGetTime()` também podem ser usadas para medir o tempo de execução e detectar atrasos. Essas funções podem ser empregadas de maneira semelhante às abordagens anteriores para avaliar o impacto da depuração no tempo de execução do código (Ferrie, 2011).

4.3 Anti-VM

As técnicas *anti-VM* têm desempenhado um papel importante na evasão de *malwares*, principalmente em ambientes de análise controlados. Uma vez que pesquisadores de segurança utilizam amplamente máquinas virtuais (*VMs*) para examinar *malwares* sem comprometer sistemas reais, os desenvolvedores de *malware* têm evoluído suas estratégias para detectar esses ambientes e mascarar seu comportamento malicioso. Portanto, *malwares* podem identificar a presença de *VMs* ao analisar assinaturas de ferramentas de virtualização como *VirtualBox* ou *VMware*, observando, por exemplo, entradas específicas no registro do *Windows* ou endereços *MAC* característicos em sistemas operacionais como o *macOS* (Singh and Singh, 2018). Essa capacidade permite que o *malware* esconda seu comportamento, frustrando o processo de análise.

O conceito de “Split Personality Malware”, introduzido por Vishnani et al. (2011), destaca uma classe de *malwares* ainda mais sofisticada, que detecta ambientes de análise e altera seu comportamento para parecer benigno. Isso força os analistas a recorrerem a sistemas nativos para capturar a verdadeira natureza maliciosa do código, aumentando o tempo e a complexidade da análise. Em resposta a essa ameaça, ferramentas como o *VMDetectGuard* (Vishnani et al., 2011), o *COBAI* (Crăciun et al., 2023) e *PinVMShield* (Rodríguez et al., 2016) foram desenvolvidas para enganar o *malware*, fazendo-o acreditar que está rodando em um ambiente nativo, mesmo quando está em uma *VM*.

De acordo com Lee et al. (2021), técnicas *anti-VM* são frequentemente empregadas por protetores comerciais para detectar quando um software está sendo executado em um ambiente de máquina virtual e, assim, encerrar sua execução para impedir a análise. Essas técnicas de detecção são classificadas por Lee et al. (2021) em três abordagens principais: baseadas em registro, onde diferenças nos valores do registro entre o sistema operacional convidado e o hospedeiro revelam o ambiente virtual; baseadas em hardware, que exploram discrepâncias nas informações de hardware entre o *SO* convidado, que utiliza hardware virtual via *hypervisor*, e o hardware físico do *SO* hospedeiro; e baseadas em processos/serviços, que identificam programas ou serviços específicos presentes apenas no *SO* convidado. Os autores observam que, embora essas técnicas *anti-VM* sejam amplamente implementadas, há uma escassez de pesquisas empíricas

explorando quais métodos são usados em protetores comerciais reais e como eles podem ser contornados.

Apesar dos avanços na virtualização, a eficácia das técnicas *anti-VM* ainda representa um desafio significativo, especialmente quando combinadas com técnicas *anti-debugging* e *anti-instrumentação*. Embora ferramentas como a *Instrumentação Binária Dinâmica (DBI)* ofereçam maneiras de contornar essas técnicas, sua aplicação em programas complexos pode ser limitada (Kim et al., 2020).

No geral, o desenvolvimento de técnicas *anti-VM* faz parte de uma estratégia mais ampla dos criadores de *malwares* para escapar da detecção e dificultar a análise. Essas técnicas são parte de uma taxonomia mais ampla de métodos *anti-análise*, que buscam explorar diferenças intrínsecas entre sistemas normais e virtualizados para evitar monitoramento (Chen et al., 2008). Ao categorizar esses métodos, podemos entender melhor e, eventualmente, mitigar seus efeitos nas análises de segurança.

A seguir, apresentamos algumas das principais categorias dessas técnicas, ilustradas com exemplos específicos citados por diversos autores.

4.3.1 Detecção Baseada em Registro (Registry-based Detection)

A detecção baseada em registro envolve a inspeção de chaves e valores específicos no registro do sistema operacional que indicam a presença de um ambiente virtual. Olaimat et al. (2021) mencionam que *malwares* podem localizar chaves de registro que identificam máquinas virtuais, como `HKLM\SOFTWARE\Vmware Inc.\Vmware Tools` para detectar o *VMware*. Além disso, eles podem procurar por *drivers* específicos como “`VMToolsHook.dll`” no caminho `C:\windows\System32\Drivers`, que está associado ao *VMware*.

Lee et al. (2021) explicam que programas protegidos com o *Themida* leem valores no registro para detectar *VMs*. Como observado na Figura 4.6, para identificar o *VMware*, o programa lê o valor “`VMware SVGA 3D`” na chave “`DriverDesc1`”. No caso do *VirtualBox*, verifica o valor “`VBOX__`” em “`SystemBiosVersion`” e verifica se a pasta “`VBOX__`” existe no registro nos caminhos `HKEY\_LOCAL\_MACHINE\HARDWARE\DESCRIPTION\System` e `HKEY\_LOCAL\_MACHINE\HARDWARE\ACPI\DSDT\VBOX\_`.

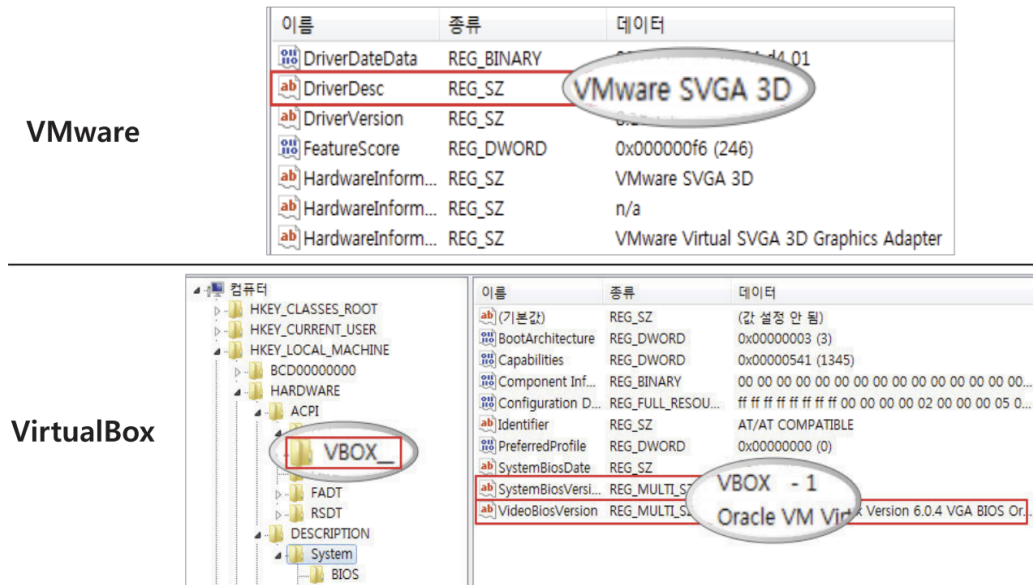


Figura 4.6: Chaves de registro usadas pelo *Themida* para detectar o *VMware* e o *VirtualBox* (Lee et al., 2021).

O *Themida* detecta o *VirtualBox* verificando as chaves de registro usando a API *RegOpenKeyExA* como observado na Figura 4.7:

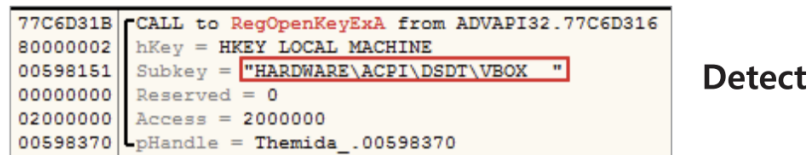


Figura 4.7: Técnica de detecção através da API *RegOpenKeyExA* (Lee et al., 2021).

Vishnani et al. (2011) observam que as entradas do registro podem conter centenas de referências à string “VMware” no sistema operacional convidado. Valores específicos no registro podem revelar a presença de uma *VM*. Alguns exemplos destacados por Vishnani et al. (2011) incluem as seguintes chaves de registro:

```

HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\Scsi\Scsi Port1\Scsi Bus 0\Target Id 0\Logical Unit Id
0\Identifier -> VMware, VMware Virtual S1.0

HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E968-E325-11CE-BFC1-08002BE10318}
\0000\DriverDesc -> VMware SCSI Controller

HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\Class\{4D36E968-E325-11CE-BFC1-08002BE10318}
\0000\ProviderName -> VMware, Inc.
    
```

4.3.2 Detecção Baseada em Hardware (Hardware-based Detection)

Nesta abordagem, o *malware* identifica discrepâncias nas informações de *hardware* entre ambientes físicos e virtuais. Olaimat et al. (2021) indicam que *malwares* podem validar parâmetros únicos do *hardware*, como números de série do *chipset*, *placa-mãe* ou *processador*, para diferenciar entre máquinas físicas e virtuais. Neste sentido, Lee et al. (2021) descrevem que ferramentas de proteção como o *VMProtect* podem utilizar informações da tabela de *firmware* obtidas via a API “*GetSystemFirmwareTable*” para detectar *VMs*. O programa realiza correspondência de *strings* nos dados da tabela de *firmware*, procurando por “VMwa” para detectar o *VMware* e “Virt” para o *VirtualBox*.

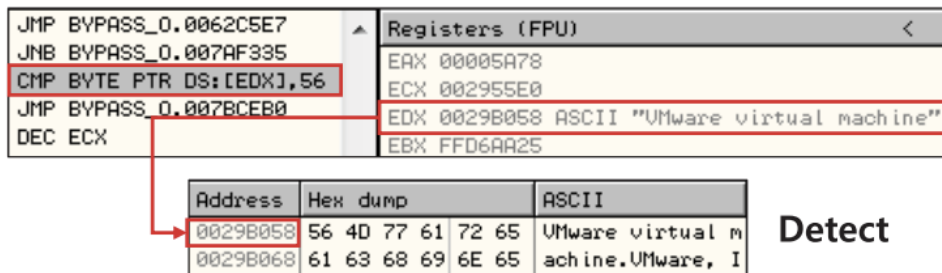


Figura 4.8: Técnica de detecção do *VMware* utilizando informações da tabela de *firmware* (Lee et al., 2021).

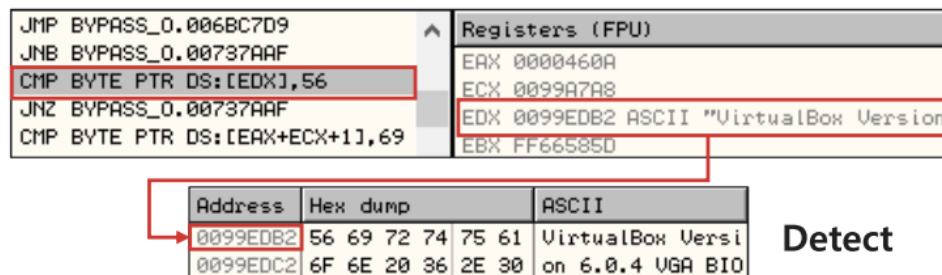


Figura 4.9: Técnica de detecção do *VirtualBox* utilizando informações da tabela de *firmware* (Lee et al., 2021).

As Figuras 4.8 e 4.9 demonstram que o *VMware* é detectado comparando as informações da tabela de *firmware* por 1 byte sequencialmente a partir do endereço 0x564D7761 (ou seja, VMwa), e o *VirtualBox* é detectado comparando 1 byte sequencialmente a partir do endereço 0x56697274 (ou seja, Virt).

Afianian et al. (2018) e Chen et al. (2008) mencionam que *malwares* podem procurar por dispositivos e *drivers* emulados por *VMs*, como adaptadores *Ethernet* com prefixos de fabricante específicos, que são indicadores de um ambiente virtual. Nesse mesmo sentido, Olaimat et al. (2021) destaca que os endereços dos adaptadores de rede virtuais revelam a identidade de um fornecedor de *VM*. Por exemplo, se o prefixo dos endereços *MAC* for igual a “00-0C-29”, eles estão associados ao *VMware*. Se o prefixo for igual a 08:00:27, eles estão associados ao *Oracle VirtualBox*.

4.3.3 Detecção de Processos e Serviços

A detecção de processos e serviços verifica a existência de processos ou serviços específicos característicos de ambientes virtuais. Olaimat et al. (2021) citam que a presença de processos como “VMwareTray” e “Vmscsi” para o *VMware*, ou “vboxtray” para o *VirtualBox*, pode indicar que o *malware* está sendo executado em uma *VM*.

Lee et al. (2021) explicam que o *Enigma* detecta o *VirtualBox* buscando pelo processo “VBoxService.exe”, que é instalado com o *VirtualBox Guest Additions*. O programa utiliza a *API Process32Next* para obter a lista de processos em execução e compara os nomes dos processos com “VBoxService.exe”, armazenando essas informações em *Unicode* na memória. Depois disso, o *Enigma* usa a *API “WideCharToMultiByte”* para comparar com o valor *ASCII* “VBoxService.exe” predefinido. A Figura 4.10 ilustra como a *API WideCharToMultiByte* converte valores *Unicode* armazenados na memória para *ASCII*, permitindo a comparação para realizar a detecção.

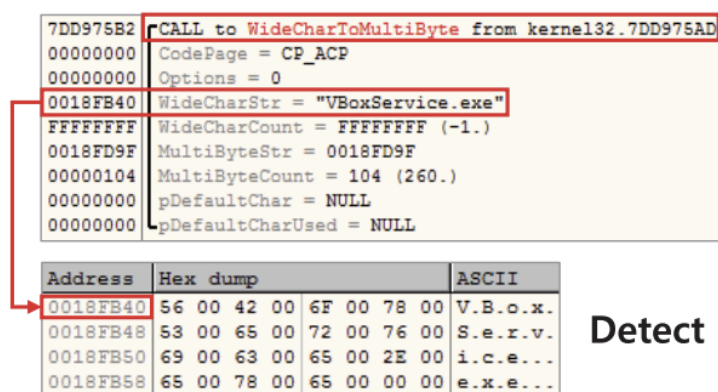


Figura 4.10: Técnica de detecção através da *API WideCharToMultiByte* (VBoxService.exe) (Lee et al., 2021).

Vishnani et al. (2011) destacam que processos específicos do *VMware*, como “VMwareUser.exe”, “VMwareService.exe”, “vmacthlp.exe” e “VMwareTray.exe”, podem ser monitorados por *malwares* para detectar a presença de uma *VM*.

4.3.4 Detecção Baseada em Sistema de Arquivos e Drivers no Sistema

A detecção baseada em arquivos e sistema de arquivos envolve a procura por arquivos específicos ou estruturas de diretórios exclusivos de ambientes virtuais. Vishnani et al. (2011) apontam que existem arquivos e pastas específicos do *VMware* que podem ser consultados para detectar o ambiente virtual, e *malwares* podem buscar por esses objetos no sistema de arquivos para confirmar a execução em uma *VM*. Sikorski and Honig (2012) menciona que o diretório de instalação do *VMware*, geralmente em C:\Program Files\VMware\VMware Tools, pode conter artefatos que indicam a presença de uma *VM* e podem ser explorados por *malwares*. Olaimat et al. (2021) mencionam que *malwares* podem explorar a presença de arquivos de instalação de sistemas virtuais, como C:\windows\System32\Drivers\VMToolsHook.dll para o *VMware* ou C:\windows\System32\Drivers\VBoxSF.sys para o *VirtualBox*.

Da mesma forma, Lee et al. (2021) explicam que o *Obsidium* detecta o *VirtualBox* verificando a existência do *driver* “VBoxGuest”. O programa utiliza a *API “CreateFileW”* com o nome do

arquivo `\\.\VBoxGuest` para verificar se ele existe; se o *driver* for encontrado, o *malware* conclui que está sendo executado no *VirtualBox*.

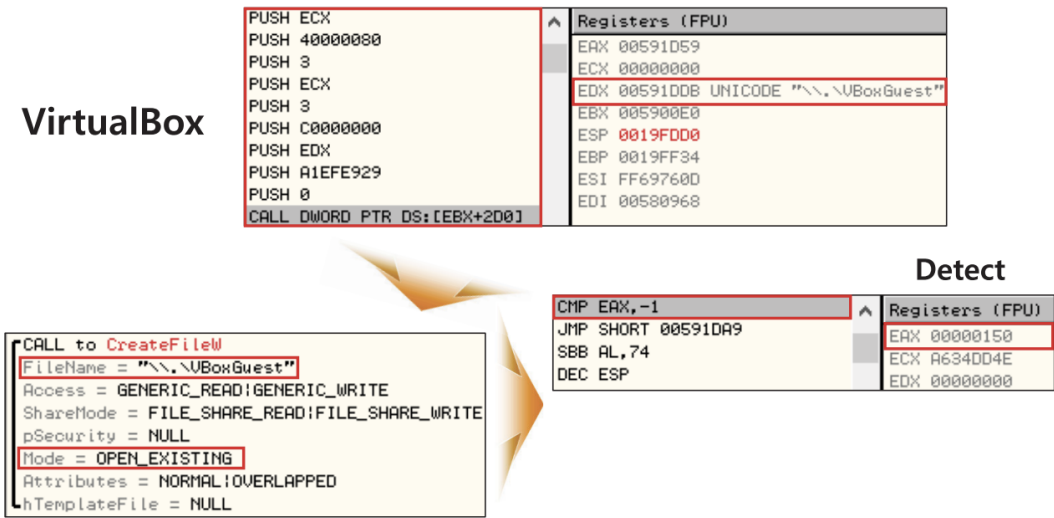


Figura 4.11: Detecção através do *driver* `VBoxGuest` (Lee et al., 2021).

Se o valor de retorno do registrador `EAX` for `-1` (`0xFFFFFFFF`) após executar a *API*, isso indica que o *driver* `VBoxGuest` não existe e que o sistema não está no *VirtualBox*. Por outro lado, se o valor do registrador `EAX` retornar com um valor diferente de `-1`, como ilustrado na Figura 4.11, isso indica que o *driver* `VBoxGuest` existe e o *malware* está sendo executado no *VirtualBox*.

4.4 Anti-Instrumentação

Esta seção aborda as estratégias usadas por softwares maliciosos para detectar e evadir ambientes de *instrumentação binária dinâmica* (*DBI*), assim como *frameworks* de análise de comportamento como o *PinVMShield* (Rodríguez et al., 2016), o *PINvader* (Gozzini et al., 2022) ou o *COBAI* (Crăciun et al., 2023). Essas técnicas são projetadas para dificultar a coleta de informações durante a execução de programas sob ferramentas como *Pin*, *DynamoRIO*, e outras (Crăciun et al., 2023). Assim, as técnicas anti-instrumentação representam uma classe sofisticada de estratégias utilizadas por *malwares* para detectar e evitar a análise por ferramentas de *DBI* (Polino et al., 2017a; Lee et al., 2021; Filho, 2019). Esses mecanismos evoluíram para detectar as características comuns de um ambiente *DBI* e explorar vulnerabilidades nas arquiteturas desses sistemas, criando desafios contínuos para pesquisadores de segurança (Park and Park, 2024).

D’Elia et al. (2019) definem evasão de *DBI* como o desvio dos fluxos de instrução e dados de um programa em relação ao comportamento esperado em uma execução nativa, causado por falhas de tradução ou pela detecção do motor *DBI*. A evasão ocorre quando o código adapta seu comportamento ao detectar a presença de um ambiente de análise, ou quando utiliza técnicas como código automodificante para gerar fluxos de controle errôneos, impedindo a análise correta.

Em contraste, fuga de *DBI* ocorre quando partes do código conseguem escapar do controle do motor *DBI* e são executadas diretamente no processador, fora do ambiente monitorado (D’Elia et al., 2019).

A dissertação de Filho (2019) explora como *malwares* modernos usam técnicas anti-instrumentação para detectar e comprometer ambientes de *DBI*, levantando preocupações sobre a segurança e eficácia dessas ferramentas na análise de comportamento malicioso. Diante do avanço das técnicas evasivas, o estudo propõe novas contramedidas que, embora não garantam transparência total, reduzem a superfície de ataque explorável.

D’Elia et al. (2019) também exploram como as ferramentas *DBI* introduzem artefatos detectáveis e propõem que uma reformulação arquitetural pode ser necessária para mitigar esses artefatos (D’Elia et al., 2019). Polino et al. (2017a) e Filho (2019) corroboram essa análise, destacando que métodos evasivos avançados frequentemente exploram o tempo de execução, o que torna a ocultação de artefatos insuficiente. A evolução das técnicas evasivas força os defensores a adotarem estratégias sofisticadas. Autores como Filho et al. (2022) e Crăciun et al. (2023) sugerem módulos especializados para mascarar alterações estruturais durante a instrumentação.

No geral, a discussão sobre técnicas anti-instrumentação deixa claro que não há uma solução única para lidar com as complexas técnicas evasivas de *malwares* modernos. A Tabela 4.1 apresenta diferentes taxonomias para classificar como os *malwares* detectam e evitam ambientes *DBI*, refletindo a diversidade de abordagens para contornar a análise (Lee et al., 2021; Polino et al., 2017a; Park and Park, 2024; D’Elia et al., 2019; Rodríguez et al., 2016).

A seguir, apresentamos alguns exemplos das principais categorias das técnicas anti-instrumentação citadas por diversos autores. Essas categorias refletem o contínuo avanço das estratégias de evasão usadas por *malwares*.

4.4.1 Detecção de Artefatos de Ambiente (*Environment Artifacts*)

Uma das categorias mais comuns e eficazes de técnicas anti-instrumentação envolve a busca por artefatos inseridos por ferramentas de *instrumentação binária dinâmica* (*DBI*). Como discutido por Filho (2019) e Polino et al. (2017a), a presença de estruturas inseridas pelo *framework Pin* é frequentemente utilizada como evidência da instrumentação. O estudo de D’Elia et al. (2019) explora falhas na ocultação de informações sensíveis deixadas pelo ambiente de instrumentação. Técnicas como a verificação do estado real do registrador *EIP* usando instruções como *fstenv* e *int 2e*, ou a inspeção de endereços de memória, revelam artefatos internos dos *frameworks DBI* que normalmente não estariam visíveis em um programa nativo.

Polino et al. (2017a), Falcón and Riva (2012) e Park and Park (2024) descrevem a técnica de verificação do processo pai (*Parent Detection*). Essa técnica explora o fato de que, em um ambiente de instrumentação, o programa instrumentado é lançado como um processo filho do instrumento *DBI*, por exemplo, o *pin.exe*, como apresentado na Figura 4.12. O *malware* verifica o processo pai para identificar se é um *DBI* em vez de um processo esperado, como *cmd.exe*.

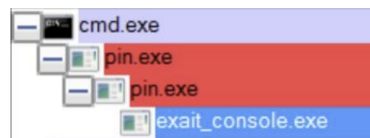


Figura 4.12: Detecção de Processo Pai (Polino et al., 2017b)

O Algoritmo 1 apresentado por Park and Park (2024) verifica o processo pai de um programa

Tabela 4.1: Técnicas de Detecção de Evasão por Autor

Autor	Técnicas
Lee et al. (2021)	• Baseada em Sobrecarga (Overhead-based) • Baseada em Compilador JIT (JIT Compiler-based) • Baseada em API (API-based) • Baseada em Exceções (Exception-based)
Polino et al. (2017a)	• Artefatos de Cache de Código (Code Cache Artifacts) • Artefatos de Ambiente (Environment Artifacts) • Detecção de Compilador JIT (JIT Compiler Detection) • Detecção de Sobrecarga (Overhead Detection)
Park and Park (2024)	• Sobrecarga Temporal (Time Overhead) • Artefato de Cache de Código (Code Cache Artifact) • Impressões Digitais de Memória (Memory Fingerprints) • Compilador JIT (JIT Compiler) • Artefato de Ambiente (Environment Artifact)
D’Elia et al. (2019)	• Sobrecarga Temporal (Time Overhead) • Vazamento de Ponteiros de Código (Leaked Code Pointers) • Conteúdo e Permissões de Memória (Memory Contents and Permissions) • Componentes Internos do DBI (DBI Engine Internals) • Interações com o Sistema Operacional (Interactions with the OS) • Manipulação de Exceções (Exception Handling) • Defeitos de Tradução (Translation Defects)
Rodríguez et al. (2016)	• Diretas – Reconhecimento de Análise de Binário – Reconhecimento de Análise em Ambiente Isolado – Reconhecimento de Análise de Memória • Indiretas – Manipulação Temporal (e.g., execução atrasada) – Execução Condicional Baseada em Eventos (e.g., atividade do usuário)

em execução para detectar a presença de ferramentas de instrumentação, como o *Pin*. O algoritmo compara o identificador de processo atual (*PID*) com o identificador de processo no *Process Entry (PE)*. Se coincidirem, ele obtém o identificador do processo pai (*ParentProcessID*) e, em seguida, utiliza a função *GetModuleFileName* para capturar o nome do módulo associado ao processo pai. Se o nome do processo pai for igual a “pin.exe” ou “drrun.exe” (indicadores comuns de ferramentas de instrumentação), a variável *detected* é definida como *True*, indicando que o programa está sendo monitorado por uma ferramenta de análise.

Algoritmo 1 Verificação do Processo Pai (Park and Park, 2024)

Data: *Process Entry (PE)*, *Current Process ID (PID)*

```

1: if PE.ProcessID == PID then
2:   pPID ← PE.ParentProcessID
3:   parentName ← GetModuleFileName(pPID)
4:   if parentName == “pin.exe” or parentName == “drrun.exe”,... then
5:     detected ← True
6:   end if
7: end if

```

Uma das técnicas anti-DBI do *Obsidium* descrita por Lee et al. (2021) utiliza a *API ZwQueryInformationProcess* para detectar a presença de ferramentas de análise e depuração. A *API* é usada para verificar se o processo está sendo monitorado, fornecendo duas opções: uma abordagem documentada com o argumento 0x7 (*ProcessDebugPort*), que detecta depuradores tradicionais, e uma abordagem não documentada com o argumento 0x1F, que é capaz de detectar ferramentas de DBI. A Figura 4.13 ilustra a execução do código utilizando a *API ZwQueryInformationProcess* com diferentes argumentos e as condições de retorno que determinam a detecção ou *bypass* do ambiente de análise.

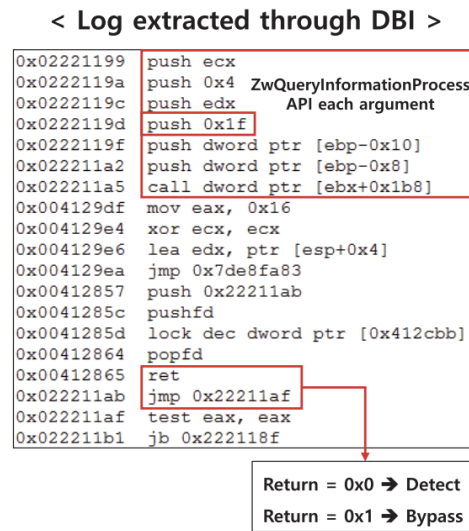


Figura 4.13: Detecção através da *API ZwQueryInformationProcess* (Lee et al., 2021).

No exemplo da Figura 4.13, cada valor de argumento é passado para a *API* em uma sequência

de instruções *assembly*. A sequência de código começa empilhando (*push*) os valores necessários antes de realizar a chamada à *API* (usando a instrução *call*). Quando o valor 0x1F é passado como argumento, a *API* retorna 0 se uma ferramenta de DBI estiver anexada ao processo (indicando “Detect”) ou 1 caso contrário (“Bypass”). Após a verificação, o programa usa a instrução *test eax, eax* para avaliar o valor de retorno e, com base nesse valor, desvia o fluxo de controle, terminando ou continuando a execução.

A técnica descrita por Lee et al. (2021) se aproveita de argumentos não documentados para inspecionar diretamente se há um depurador ou instrumentador monitorando o processo, caracterizando uma verificação ativa de indicadores deixados por essas ferramentas no ambiente de execução.

Esse tipo de técnica é eficaz porque explora uma funcionalidade pouco documentada da *API*, tornando a detecção de DBIs mais difícil para analistas que utilizam métodos convencionais de evasão de depuração. Assim, mesmo que um analista tente contornar o comportamento de detecção usando valores conhecidos, a verificação com o argumento 0x1F pode identificar com precisão a presença de instrumentadores e reagir de acordo, interrompendo a análise.

Semelhante ao uso da *API ZwQueryInformationProcess* para verificar a presença de depuradores conforme descrito por Lee et al. (2021), D’Elia et al. (2019) discutem métodos que revelam o DBI por meio de vazamento de ponteiros reais ou funções exportadas, demonstrando que existem muitas outras abordagens que exploram características expostas dos DBIs para detectar seu uso e evitar a análise.

4.4.2 Baseada em Exceções (*Exception-based*)

Lee et al. (2021) e D’Elia et al. (2019) descrevem as técnicas baseadas em exceções (*exception-based*) como métodos de evasão que exploram a incapacidade das ferramentas de *instrumentação binária dinâmica* (DBI) de gerenciar corretamente exceções geradas durante a execução. Se um programa manipula uma exceção com uma rotina específica e essa rotina não é acionada pelo DBI, isso indica a presença de um ambiente de análise, permitindo que o *malware* detecte a ferramenta e interrompa sua execução (Lee et al., 2021).

Lee et al. (2021) apresentam um exemplo de técnica evasiva baseada em exceções onde os autores abordam uma detecção que explora o uso de exceções do tipo “single step”, utilizada pelo protetor *VMProtect*. A técnica utiliza a modificação do comportamento de exceções e flags no registrador *EFLAGS* para desviar o fluxo normal de execução, alterando como o programa se comporta em ambientes de análise. Embora o *VMProtect* não tenha uma opção de anti-DBI explícita, ele utiliza estratégias anti-debug que, indiretamente, também dificultam a análise por ferramentas de DBI. A estratégia baseia-se no uso da flag de *trap* (*trap flag*), configurada por meio do valor 0x100 no registrador *EFLAGS*, ativada por meio da instrução *POPF*. Quando a flag é ativada, ela força o ambiente de execução a entrar em um estado de *single step*, onde cada instrução subsequente é executada uma por vez, gerando uma exceção que pode ser manipulada.

Na presença de um ambiente de análise como um *debugger* ou uma ferramenta DBI, essa manipulação não ocorre como esperado, levando a um comportamento anômalo que resulta na terminação do programa. O exemplo mostrado na Figura 4.14 representa esse processo: ao executar a instrução *POPF* com o *trap flag* configurado, o valor 0x100 é empilhado. Em um ambiente DBI, esse comportamento provoca um estado inconsistente, o que é detectado e usado para interromper a execução.

Portanto, essa técnica visa detectar e reagir a anomalias no controle de fluxo, induzindo falhas em depuradores e ferramentas de DBI, o que caracteriza uma forma de manipulação intencional do fluxo de execução para fins de evasão.

Lee et al. (2021) também demonstra a contramedida que envolve a detecção prévia desse *trap*

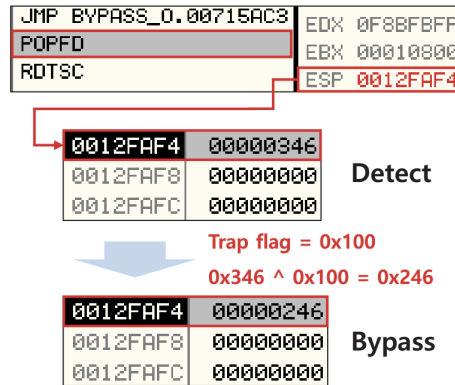


Figura 4.14: Mitigando a técnica de detecção por uma operação XOR com trap flag (Lee et al., 2021).

flag na pilha e a realização de uma operação *XOR* para removê-lo antes da execução de **POPFD** Desempilha Registrador de Flags (*Pop Flags Register*). A lógica apresentada na Figura 4.14 ilustra o processo, onde o valor original 0x346 (com o *trap flag* habilitado) é modificado para 0x246, desativando o *trap flag* e evitando a geração da exceção. Dessa forma, o comportamento evasivo não é ativado, permitindo que o programa opere normalmente mesmo sob um ambiente de instrumentação. Esse método permite que analistas desativem a proteção sem modificar o fluxo principal do programa, garantindo assim que a análise prossiga sem interrupções.

4.4.3 Conteúdo e Permissões de Memória (*Memory Contents and Permissions*)

Ferramentas de Instrumentação Binária Dinâmica (**DBI**), como o *Pin*, *DynamoRIO* e *Frida*, deixam artefatos durante a execução (Polino et al., 2017a; Park and Park, 2024; Rodríguez et al., 2016), como modificações em permissões de memória, inconsistências no código cacheado ou mudanças na estrutura de execução (Polino et al., 2017a). *Malwares* sofisticados podem detectar esses artefatos ao verificar alterações no comportamento de execução do código. Isso inclui o monitoramento de permissões de páginas de memória, onde malwares detectam modificações introduzidas por *frameworks DBI*, como páginas marcadas com permissões **RWX** (leitura, escrita e execução) (Park and Park, 2024; Polino et al., 2017a; D’Elia et al., 2019; Filho, 2019; Lee et al., 2021; Crăciun et al., 2023).

A principal técnica baseada em Permissões de Memória chamada de “RWX Permission” (Park and Park, 2024), descrita pelos autores Park and Park (2024); Polino et al. (2017a); D’Elia et al. (2019); Filho (2019); Lee et al. (2021); Crăciun et al. (2023), explora uma característica específica de motores **JIT** Compilação Just-In-Time (*Just-In-Time Compilation*) em *frameworks* de instrumentação, que é a criação de páginas de memória com permissões de leitura, escrita e execução (RWX). Motores JIT utilizam essas páginas para armazenar instruções compiladas dinamicamente, o que é incomum em programas comuns, onde páginas com permissões RWX são raras por razões de segurança e proteção de memória.

O Algoritmo 2 verifica a presença dessas permissões de memória incomuns. Utilizando a função *VirtualQueryEx*, que analisa as permissões de memória de um processo especificado, o código percorre todas as regiões de memória do processo (representadas pela estrutura *Memory Basic Information Structure, mbi*) e contabiliza as páginas com permissões

Algoritmo 2 RWX Permission (Park and Park, 2024)

Data: Identificador de Processo Atual (*Current Process Handle*, $hProcess$), Estrutura de Informação Básica de Memória (*Memory Basic Information Structure*, mbi)

```

1: while VirtualQueryEx( $hProcess$ ,  $addr$ , addressof( $mbi$ ), ...) do
2:   if  $mbi.Protect == PAGE\_EXECUTE\_READWRITE$  then
3:      $rwxCOUNT = rwxCount + 1$ 
4:   end if
5:    $addr = addr + N$ 
6: end while
7: if  $rwxCOUNT > n$  then
8:    $detected \leftarrow \text{True}$ 
9: end if

```

PAGE_EXECUTE_READWRITE. Quando o número de páginas com permissões RWX ($rwxCOUNT$) excede um determinado valor (n), isso indica a presença de um motor JIT, como um DBI, permitindo assim detectar a instrumentação.

4.4.4 Medidas Temporais (*Time Overhead*)

Lee et al. (2021) classificam as medidas temporais como uma das principais categorias de técnicas anti-DBI, uma vez que o impacto no tempo de execução pode ser medido por meio de contadores de ciclo de CPU ou funções de temporização, permitindo ao *malware* detectar se está sendo monitorado e terminar seu processo. De forma semelhante, Polino et al. (2017a) afirma que a detecção de *overhead* é particularmente eficaz para identificar instrumentos DBI, pois a latência adicional introduzida pelas ferramentas pode ser facilmente observada comparando o tempo de execução de diferentes blocos de código. Em cenários práticos, esse método é utilizado para medir a latência de instruções críticas, *loops* ou chamadas a funções que, em um ambiente instrumentado, apresentam tempos de resposta significativamente maiores. *Malwares* sofisticados podem explorar essa característica como um indicador da presença de ferramentas de análise, alterando seu comportamento para evitar detecção.

(Park and Park, 2024) também discutem a técnica de sobrecarga temporal como uma abordagem válida para detectar *frameworks* como o *Intel Pin*. Eles observam que a diferença na carga e na execução de instruções, especialmente durante a tradução Just-In-Time (JIT), pode resultar em flutuações que não são vistas em execuções nativas. Isso torna a sobrecarga de tempo uma medida confiável para identificar a presença de DBIs. (D'Elia et al., 2019) ampliam essa análise ao explorar como o uso de código automodificante e a repetição de instruções no cache podem gerar padrões de tempo anômalos, mesmo com mecanismos avançados de otimização. Segundo os autores, um exemplo típico é a observação do tempo necessário para carregar bibliotecas dinamicamente, que pode ser duas ordens de magnitude maior em um ambiente instrumentado.

D'Elia et al. (2019) abordam as medidas temporais como uma das principais limitações de ferramentas de DBI, pois a sobrecarga temporal introduzida pela instrumentação não pode ser completamente eliminada. Os autores demonstram que a medição de tempos de carga de bibliotecas e execuções de funções pode variar de forma significativa em ambientes instrumentados, permitindo ao *malware* detectar facilmente a presença de um *framework DBI* como *Pin* ou *DynamoRIO*. Crăciun et al. (2023) adicionam que a falta de virtualização completa em muitos *frameworks DBI* pode amplificar ainda mais essa sobrecarga, gerando padrões temporais

anômalos que *malwares* sofisticados conseguem identificar.

Crăciun et al. (2023) também afirmam que aplicações maliciosas geralmente exploram os limites da taxa de virtualização (aspectos de virtualização conhecidos não cobertos ou problemas de virtualização desconhecidos do público) ou o aumento do *overhead* de recursos (tempo, CPU, memória) causado pela presença de um DBI.

Um exemplo prático de *overhead* causado por instrumentação e como ele pode ser detectado usando a API Interface de Programação de Aplicações (*Application Programming Interface*) *QueryPerformanceCounter* é abordado por Polino et al. (2017a). Eles descrevem como *malwares* podem usar funções de temporização do Windows para medir o impacto temporal gerado pela instrumentação e, assim, detectar a presença de ferramentas de análise. Para isso, o *malware* pode executar pequenos testes de desempenho e comparar os tempos de execução obtidos com tempos esperados em uma execução nativa. Caso seja identificada uma diferença significativa, isso indicaria a presença de sobrecarga (*overhead*), sugerindo que o programa está sendo monitorado.

Algoritmo 3 Medição de Tempo de Execução (Park and Park, 2024)

Data: Pointer to a Counter Value (*ticks1*, *ticks2*)

```

1: QueryPerformanceCounter(&ticks1)
2: QueryPerformanceCounter(&ticks2)
3: if (ticks1.QuadPart - ticks2.QuadPart) ≤ n then
4:   detected ← True
5: end if
```

O exemplo de algoritmo fornecido por Park and Park (2024), ilustrado no Algoritmo 3, utiliza a função **QueryPerformanceCounter** Contador de Desempenho de Consultas para medir o tempo de execução de um bloco de código e detectar a presença de sobrecarga causada por ferramentas de DBI. O algoritmo proposto, denominado **Execution Time** Tempo de Execução, realiza duas chamadas consecutivas à função *QueryPerformanceCounter*, armazenando os valores de contagem de ciclos (**ticks**) em **ticks1** e **ticks2**, nas variáveis correspondentes. A diferença entre os valores dos campos *QuadPart* de *ticks1* e *ticks2* representa o tempo decorrido entre as chamadas. Caso essa diferença seja inferior a um limiar predefinido **n**, a técnica considera que há uma sobrecarga anômala, indicando a possível presença de um ambiente de instrumentação binária.

Segundo Park and Park (2024), o uso de *QueryPerformanceCounter* é ideal para esse tipo de medição, pois oferece maior precisão e resolução em comparação com outras APIs do Windows, facilitando a detecção de variações sutis no tempo de execução.

Portanto, as medidas temporais representam um conjunto de técnicas evasivas eficientes para detectar ambientes de instrumentação por meio de monitoramento de performance, exigindo dos pesquisadores a adoção de estratégias de mitigação, como a otimização de instruções ou a redução de verificações para minimizar a sobrecarga e impedir que o tempo de execução seja um indicador confiável da presença de DBIs.

4.4.5 Outras Técnicas

Lee et al. (2021) identificam categorias como técnicas baseadas em compiladores **JIT** Compilação Just-In-Time (*Just-In-Time Compilation*) e baseadas em APIs (API-based), que utilizam artefatos específicos do funcionamento interno dos *frameworks* para detectar instrumentação, enquanto Polino et al. (2017a) destaca a presença de artefatos de cache de código (**Code Cache Artifacts**) Artefatos de Cache de Código e detecção de compiladores JIT (**JIT Compiler**

Detection) Detecção de Compiladores JIT como pontos fracos que podem ser explorados para identificar inconsistências geradas pelos DBIs. [Park and Park \(2024\)](#) enfatizam técnicas como impressões digitais de memória (**Memory Fingerprints**) e artefatos de ambiente (**Environment Artifacts**) para revelar a presença de DBIs por meio de características no layout de memória e comportamento de execução, utilizando métodos como verificação de duplicação de argumentos na memória e detecção de diferenças de latência de compilação.

Por outro lado, [D’Elia et al. \(2019\)](#) oferecem uma taxonomia mais detalhada, incluindo a categoria de defeitos de tradução (**Translation Defects**), que explora falhas inerentes na tradução binária dos *frameworks*, como instruções de código automodificante (**SMC** Código Automodificante - *Self-Modifying Code*) que não são tratadas corretamente e técnicas como o “Heaven’s Gate”, que explora a troca entre segmentos de 32-bit e 64-bit para escapar de instrumentação. Já [Rodríguez et al. \(2016\)](#) propõem uma divisão separada em duas categorias principais diferenciando em técnicas diretas e indiretas, considerando se o *malware* introduz mecanismos específicos de evasão ou manipula temporizadores e eventos para alterar o comportamento durante a análise.

4.5 Discussão

Neste capítulo foram apresentadas as estratégias utilizadas por softwares maliciosos para detectar e evadir ambientes e ferramentas de análise de *malware*. As diferentes classificações mostraram que as técnicas evasivas modernas exploram uma ampla gama de vetores, desde manipulações de baixo nível, como permissões de memória e ponteiros de código, até abordagens avançadas de controle de fluxo e uso de APIs para criar comportamentos anômalos que dificultam a análise. Essa diversidade de técnicas torna cada vez mais desafiador o desenvolvimento de contramedidas eficazes, exigindo soluções de mitigação mais sofisticadas e adaptativas.

O próximo capítulo apresenta a ferramenta de análise de executáveis no formato PE para sistemas Windows *Contradef*, desenvolvida sobre o framework Intel Pin. Também abordará como otimizar a detecção e mitigação de algumas das técnicas discutidas por meio da implementação de mecanismos de identificação de ferramentas de proteção, como **Themida** e **Obsidium**, permitindo uma delimitação mais precisa do escopo de análise e uma mitigação mais eficaz contra esses comportamentos evasivos.

Capítulo 5

Contradef

Este capítulo apresenta a ferramenta *Contradef*, desenvolvida sobre o framework de instrumentação binária dinâmica Intel Pin. A ferramenta é voltada à análise de executáveis no formato PE para sistemas Windows 10 ou superiores, na arquitetura x64, com foco na análise dinâmica de *Malwares*, assim como na identificação e mitigação de técnicas evasivas comumente empregadas por *Malwares*. A *Contradef* é capaz de detectar padrões associados a mecanismos de proteção, como os utilizados por ferramentas como *Obsidium* e *Themida*, além de aplicar contramedidas para neutralizá-los. Sua arquitetura modular permite o rastreamento contínuo e configurável de instruções e eventos do sistema por meio de *callbacks* especializados, possibilitando a coleta sistemática de evidências e comportamentos suspeitos ao longo da execução.

5.1 Visão Geral da Ferramenta

A literatura recente sobre análise de *malware* baseada em DBI tende a concentrar-se em duas frentes principais: (i) mitigar técnicas evasivas (*anti-VM*, *anti-debug*, *anti-DBI*) para manter a execução e (ii) desempacotar/recuperar o código protegido, com foco em reconstrução de OEP e IAT. Embora essenciais, tais abordagens frequentemente encerram-se na etapa de *bypass/unpacking*, oferecendo menos suporte à inspeção sistemática do comportamento subsequente do binário.

A **Contradef**¹ foi concebida justamente para ultrapassar esse limite. Construída sobre o *Intel Pin*, ela não apenas neutraliza rotinas evasivas específicas observadas em protetores comerciais como *Obsidium* e *Themida*, como também fornece um arcabouço de *análise observacional* de alta granularidade, orientado à relação temporal de eventos. Em termos práticos, a ferramenta combina quatro perspectivas complementares: (i) *interceptação de APIs* sensíveis (p. ex., `NtQueryInformationProcess`, `CheckRemoteDebuggerPresent`, `IsDebuggerPresent`); (ii) *rastreamento seletivo* de funções, memória e instruções, ativado sob gatilhos de interesse; (iii) *contramedidas* para preservar a execução quando desejado, ou registro explícito das sondagens como indícios de evasão quando a transparência é a prioridade analítica; e (iv) **manipulação seletiva do fluxo de execução** por meio do *Detector de Sequências de Instruções* (DSI), que reconhece sequências de instruções apontadas pelo analista e aplica intervenções pontuais, possibilitando manipulações do fluxo, como redirecionamento de saltos, manipulação de `RFLAGS/EFLAGS` ou substituição controlada de destinos, permitindo forçar ramificações normalmente inacessíveis

¹<https://github.com/contradef/Contradef>

ou neutralizar técnicas de evasão sem perturbar o restante da execução.

Os módulos da **Contradef** produzem saídas em texto estruturado, favorecendo tanto a correspondência cruzada durante a execução quanto a análise posterior por scripts externos. A ferramenta inclui *hooks* especializados e heurísticas para evidenciar padrões típicos de evasão (p. ex., leituras de SMBIOS/ACPI, enumeração de janelas via `FindWindowA`, sondagens de artefatos de instrumentação ou latência via `GetTickCount`), além de mecanismos de manipulação de execução (ajuste de `flags`, parâmetros e valores de retorno) quando se deseja observar ramificações condicionadas ao ambiente.

Em contraste com soluções centradas exclusivamente em mitigação ou *unpacking*, a **Contradef** foi projetada como uma plataforma de *análise contínua*: após neutralizar a evasão, ela mantém a coleta de artefatos, como chamadas de funções, acessos à memória e instruções executadas.

A versão atual é compatível com executáveis nativos x64 no formato *Portable Executable* em Windows 10 ou superior. Binários PE de 32 bits possuem suporte parcial, e artefatos baseados em *.NET*, *Java* ou linguagens interpretadas (p. ex., PowerShell, JavaScript, VBA) não são alvo desta versão.

5.2 Implementação da Contradef

A *Contradef* foi desenvolvida em *C++* utilizando o *Visual Studio 2022* no *Windows 11 Home*, versão 23H2. Para sua implementação, foram usadas a versão 3.31 do framework *Intel Pin* e foi integrada com a versão 4.5.2 da biblioteca *YARA*².

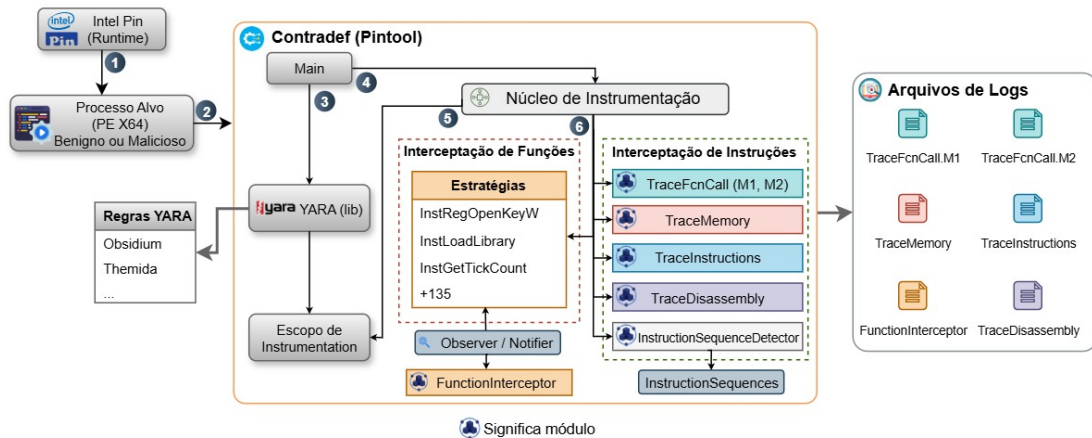
5.3 Arquitetura

A arquitetura da **Contradef** (Figura 5.1) foi projetada com foco na análise de binários protegidos por protetores como *Obsidium* e *Themida*, que incorporam técnicas anti-análise. Sua estrutura modular é composta por seis módulos especializados, capazes de operar de forma autônoma ou coordenada, com ativação seletiva conforme a estratégia de análise ou os indícios detectados. Cada módulo da arquitetura foi projetado para impulsionar a detecção de indícios de evasão. Essa modularidade permite rastrear instruções, interceptar chamadas a funções sensíveis do sistema operacional e inspecionar dados de memória em busca de indícios de evasão de diferentes tipos — *anti-VM*, *anti-debug*, *anti-disassembly* e *anti-DBI* — e aplicar as contramedidas adequadas. O processo de análise tem início com a execução de um binário no formato PE (x64), que pode ser benigno ou malicioso. Esse binário é interceptado pelo *runtime* do *Pin*, que injeta a *pintool* da *Contradef*.

O fluxo de operação da ferramenta inicia-se com o *Intel Pin Runtime* carregando o processo alvo (Passo 1), seguido pela injeção da *pintool* **Contradef** (Passo 2), que assume o controle da instrumentação antes da execução da primeira instrução do binário. O ponto de entrada da ferramenta é a função `Main`, que coordena os preparativos iniciais de análise.

Conforme ilustrado na Figura ??, a função `Main` invoca primeiramente a biblioteca *YARA* (Passo 3), realizando uma varredura estática no binário em busca de assinaturas de empacotadores como *Obsidium* e *Themida*. O resultado desta varredura estabelece o **Escopo de Instrumentação**, permitindo definir preventivamente quais módulos serão necessários. Na sequência, a função `Main` carrega o *Núcleo de Instrumentação* (Passo 4), que absorve as diretrizes do escopo para ajustar dinamicamente as configurações da análise (Passo 5) — definindo quais DLLs, APIs críticas e instruções serão monitoradas de acordo com a proteção detectada.

²<https://github.com/virustotal/yara/releases>

Figura 5.1: Arquitetura da Ferramenta *Contradef*

Com o ambiente configurado, o *Núcleo de Instrumentação* passa a coordenar os módulos (Passo 6) que operam através de dois mecanismos fundamentais de coleta de dados:

- **Interceptação de Funções:** Este mecanismo, delimitado pela caixa tracejada laranja no diagrama da arquitetura, concentra-se na captura de chamadas a funções sensíveis da *Windows API*. Sua operação fundamenta-se em um conjunto de **Estratégias** que vinculam funções críticas — tais como `InstRegOpenKeyW`, `InstLoadLibrary` e `InstGetTickCount` — a manipuladores especializados, permitindo o monitoramento seletivo de mais de 135 rotinas. A arquitetura é estruturada sobre o padrão de projeto *Observer/Notifier*, implementando um modelo de comunicação *publish/subscribe*. Nesse fluxo, o componente *FunctionInterceptor* atua como o publicador, despachando notificações detalhadas sobre os eventos interceptados, incluindo parâmetros de entrada e valores de retorno, para os módulos observadores encarregados do registro e análise dos dados.
- **Interceptação de Instruções:** Voltado para a análise de baixo nível (delimitado pela caixa tracejada verde), este mecanismo gerencia módulos como o *TraceFcnCall* (em seus modos M1 e M2), *TraceMemory*, *TraceInstructions*, *TraceDisassembly* e o *InstructionSequenceDetector*. Estes componentes rastreiam, em tempo real, desde operações de memória e fluxos de código desmontado até sequências específicas de instruções (*InstructionSequences*), permitindo uma visibilidade granular sobre a execução do binário.

Por fim, todos os dados coletados por ambos os mecanismos são direcionados para os **Arquivos de Logs**. Cada módulo gera seu próprio registro, permitindo ao analista realizar a reconstrução comportamental e a identificação de artefatos maliciosos ou técnicas de evasão empregadas pela amostra.

5.3.1 Análise Adaptativa com Detecção de Ferramentas de Proteção

Com o objetivo de tornar a instrumentação mais eficiente e adaptativa, as estratégias de análise foram organizadas com base nas ferramentas de proteção detectadas, como o *Themida* e o *Obsidium*. Essa abordagem permite delimitar dinamicamente o escopo da instrumentação e reduzir o tempo de execução, ativando apenas as verificações e contramedidas relevantes ao

tipo de proteção identificado. Ao segmentar o processo dessa forma, a análise torna-se mais direcionada, precisa e com menor sobrecarga.

Para viabilizar essa estratégia, foi necessário implementar métodos capazes não apenas de mitigar técnicas evasivas, mas também de identificar automaticamente os protetores utilizados pelos executáveis analisados. Embora ferramentas de proteção recorram a técnicas de empacotamento, ofuscação e encriptação de código, ainda é possível identificar padrões característicos por meio de análise estática. Elementos como a quantidade e os nomes das seções do executável, além de padrões sintáticos presentes nos subprogramas de ofuscação ou nos trechos responsáveis por descritografia em tempo de execução, podem ser usados como critérios de detecção.

Para incorporar essa capacidade à ferramenta, a *Contradef* integra, diretamente junto ao framework *Intel Pin*, a biblioteca do *YARA*, escrita em linguagem *C*. Com essa integração, foi possível expandir as funcionalidades da ferramenta, adicionando uma etapa preliminar de análise estática antes da execução da instrumentação propriamente dita.

A utilização do *YARA* permite detectar, ainda durante o carregamento do binário, assinaturas específicas associadas aos protetores *Themida* e *Obsidium*. Uma vez identificada a ferramenta de proteção, a *Contradef* configura dinamicamente os módulos de instrumentação, selecionando apenas as contramedidas aplicáveis. Essa etapa de pré-análise estática contribui significativamente para a redução do escopo e do custo computacional da instrumentação, além de aumentar a eficácia da análise, evitando a aplicação de técnicas genéricas desnecessárias.

Embora o *YARA* forneça regras de detecção para o *Themida* e *Obsidium*, essas regras geralmente correspondem a versões específicas que estão desatualizadas ou descontinuadas, por tanto, não funcionam nas versões mais recentes desses protetores. No entanto, como o *YARA* permite a criação de regras customizadas para detecção de padrões em executáveis, desenvolvemos assinaturas específicas voltadas à identificação das ferramentas de proteção. No escopo deste trabalho, foram implementadas regras dedicadas à detecção das versões mais recentes do *Obsidium* e *Themida*, que são os protetores objeto de estudo nesta pesquisa.

Apesar de o *Obsidium* e o *Themida* empregarem técnicas avançadas de anti-desmontagem e ofuscação — que dificultam a interpretação precisa das instruções em *assembly* por ferramentas de engenharia reversa como o *IDA Pro* —, o decompilador do *Ghidra*³ demonstrou capacidade significativa de reconstruir trechos relevantes do código em uma forma de alto nível conhecida como código pseudo-C⁴. Os recursos *CodeBrowser* e *Version Tracking*, oferecidos pelo *Ghidra*, foram especialmente úteis durante o processo de engenharia reversa, contribuindo diretamente para a definição precisa das regras *YARA* utilizadas na ferramenta.

5.3.1.1 Regra YARA para o Obsidium

Ao analisar o código decompilado e inspecionar a função localizada no *entry point* do executável com o *Ghidra*, encontramos uma função complexa que realiza operações relacionadas à proteção de dados, manipulação binária e, possivelmente, criptografia. A função faz parte de um mecanismo de proteção projetado para ocultar ou transformar dados a fim de dificultar a análise ou a engenharia reversa.

Analisamos aplicações protegidas por quatro versões do *Obsidium Demo*: 1.4.5 *build* 31, 1.7.3 *build* 5, 1.8.1 *build* 4 e 1.8.5 *build* 4. Essas versões apresentam variações diferentes da mesma função, provavelmente devido aos mecanismos internos de ofuscação aplicados durante a geração do executável protegido.

³<https://github.com/NationalSecurityAgency/ghidra/releases>

⁴Código pseudo-C é uma representação intermediária gerada por decompiladores que se assemelha à linguagem *C*, facilitando a leitura e a análise do comportamento do programa, embora nem sempre reflita com precisão o código-fonte original.

Por meio dessa análise, identificamos diversos padrões recorrentes em *assembly*, incluindo sequências específicas de bytes, que foram utilizadas para caracterizar e detectar executáveis protegidos com o *Obsidium*, conforme ilustrado nas Figuras 5.2 e 5.3.

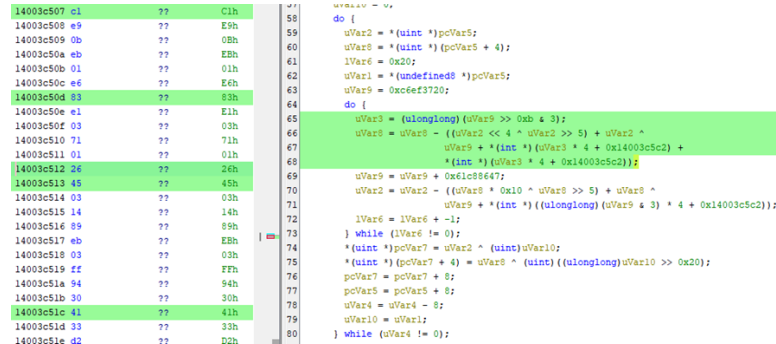


Figura 5.2: Padrão de sequência de instruções do Obsidium v1.7.3 build 5 decompilada pelo Ghidra

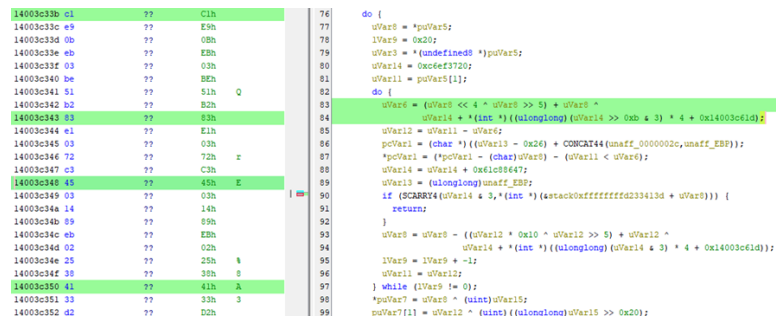


Figura 5.3: Padrão de sequência de instruções do Obsidium v1.8.5 build 4 decompilada pelo Ghidra

Com base nesses padrões e outras características estruturais, foi possível criar uma regra YARA para a identificação de executáveis protegidos por diferentes versões do *Obsidium Demo*, conforme apresentado na Listagem 5.1.

Listagem 5.1: Regra YARA para a detecção do Obsidium

```
1 import "pe"
2 rule ObsidiumDemoX64Cdf
3 {
4 meta:
5 author="Henrique Borges"
6 date_create="15/10/2024"
7 last_edit="18/10/2024"
8 description="Obsidium"
9 strings:
10 $a1 = { C1 E9 0B EB 07 [1-3] 83 E1 03 [2-5] 45 03 14 89 EB [2-5] 41 33 D2 }
11 $a2 = { 69 D2 0F 00 FF FF EB 07 }
12 $a3 = { 69 d2 0f 00 ff ff eb 07 }
13 $a4 = { 41 B8 20 00 00 00 EB }
14 $a5 = { 4C 0F 47 CB EB 07 }
15 $a6 = { 4D 2B D9 EB 07 }
16 $a7 = { 49 83 E9 01 EB }
17 $a8 = { 44 03 C2 EB 07 }
```

```

18 $a9 = { 45 03 14 89 EB }
19 $a10 = { 48 83 E3 F8 }
20 $a11 = { 45 33 DF }
21 condition :
22 pe.imports("kernel32.dll") == 1 and
23 pe.imports("kernel32.dll", "GetModuleHandleA") and
24 all of ($a*)
25 }

```

A condição final (linha 24) exige a presença de todas as *strings* de \$a1 a \$a11 no binário analisado, além da verificação de importações.

5.3.1.2 Regra YARA para o Themida

De forma análoga ao *Obsidium*, a análise do *Themida* nas versões 3.0.4, 3.1.3 e 3.1.8, revelou funções no *entry point* com instruções características de empacotadores, incluindo sequências de operações de pilha, instruções de sincronização (LOCK/UNLOCK) e manipulação de registradores. A Figura 5.4 mostra um trecho de código decompilado pelo *Ghidra*, onde esses padrões ficam evidentes.

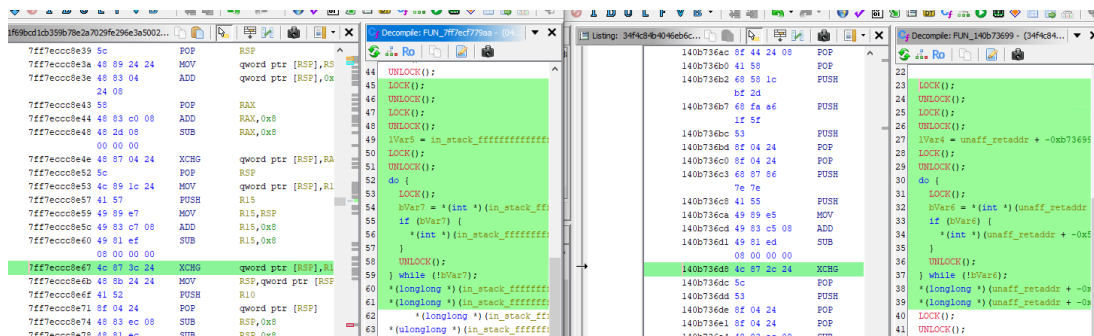


Figura 5.4: Padrão de sequência de instruções do Themida, comparando diferentes versões no *Ghidra*

Com base nesses padrões recorrentes, foi possível desenvolver uma regra *YARA* específica para detectar binários protegidos por versões mais recentes (3.XX) do *Themida*, conforme mostrado na Listagem 5.2.

Listagem 5.2: Regra YARA para a detecção do Themida 3.xx

```

1 rule Themida3X64Cdf : Packer
2 {
3   meta:
4     author="Henrique Borges"
5     date_create="24/06/2025"
6     last_edit="24/06/2025"
7     description="Themida 3.XX"
8   strings:
9     $a1 = { 58 53 51 52 56 57 55 48 89 C3 48 83 EB 05 B9 58 ?? ?? 00 48 29 CB 50 }
10    $a2 = { 48 01 D8 5D 5F 5E 5A 59 5B FF E0 }
11    $a3 = { 62 9F 55 C5 }
12    $a4 = { 52 69 63 68 26 FE 3B 96 }
13    $b1 = { 47 87 04 24 5C 53 8F 04 24 8F 04 24 FF 74 24 20 FF 34 24 }
14    $b2 = { 48 81 E9 }
15    $b3 = { 48 87 ?? 24 5C }
16    $b4 = { 31 C0 F0 0F B1 0C 2B 0F 84 07 00 00 00 F3 90 E9 EC FF FF FF FF 34 24 }

```

```

17 condition :
18 all of ($a*) or
19 all of ($b*)
20 }

```

Essa regra foi estruturada para detectar tanto padrões do tipo \$a* quanto \$b*, permitindo maior abrangência diante das diferentes variantes geradas pela ferramenta. Os padrões capturam instruções típicas do *Themida*, como empilhamento/dempilhamento de registradores, operações de restauração de estado e instruções de controle de fluxo.

As seções seguintes descrevem, em detalhe, os módulos de instrumentação disponíveis na versão atual da ferramenta.

5.3.2 Módulos da *Contradef*

5.3.2.1 Intercepção de Funções Críticas - *FunctionInterceptor*

A análise de código integrada à *Contradef* concentra-se na observação das funções e APIs específicas invocadas pelo *malware*, permitindo inferir suas funcionalidades e intenções. Essa abordagem prioriza o monitoramento de eventos do sistema, como consultas ao Registro do Windows, obtenção de informações sobre hardware, processos ou *threads*. Tais eventos, capturados em tempo de execução, fornecem evidências concretas de atividades potencialmente evasivas executadas pelo *malware* protegido, contribuindo para a compreensão do seu comportamento e para a detecção de técnicas voltadas à ocultação e anti-análise.

O módulo *FunctionInterceptor*⁵ tem a capacidade de interceptar e monitorar chamadas de sistema em tempo real, permitindo uma análise mais aprofundada dos comportamentos do *malware* enquanto ele interage com o ambiente e detecta tentativas de evitar a detecção em diferentes contextos.

Para mitigar as técnicas evasivas, o *FunctionInterceptor* pode simular condições de execução graças à capacidade do Pin de implementar ganchos (*hooks*) e interceptores antes e depois das chamadas às funções *API* do *Windows*, permitindo também a manipulação de dados em memória para modificar as respostas de funções *API* e simular diferentes condições, como a ausência de máquinas virtuais.

Para gerenciar as diferentes estratégias de intercepção de funções da API do Windows, cada método de instrumentação e eventual contramedida foi implementado seguindo o padrão de projeto comportamental conhecido como *Strategy*. Esse padrão permite definir uma família de algoritmos, encapsulá-los em classes distintas e torná-los intercambiáveis de forma transparente dentro de um mesmo contexto Shvets (2023). Com isso, a *Contradef* pode alterar dinamicamente a lógica de intercepção de acordo com a função monitorada, sem que o código cliente — isto é, os módulos que consomem os eventos — precise conhecer a implementação específica de cada estratégia. Essa abordagem facilita a extensão da ferramenta com novos mecanismos de evasão ou empacotadores, promovendo modularidade e reutilização de código.

Para cada chamada interceptada (Figura 5.5), são registrados o identificador da *thread* (*Thread Id*), o contador de chamadas (*Call Id*), o endereço absoluto da rotina, os parâmetros decodificados e o valor de retorno, independentemente de a função ter sido importada estaticamente ou resolvida em tempo de execução, característica comum em binários ofuscados.

Esse módulo se destaca pela capacidade de extrair e manipular (quando necessário), em tempo de execução, tanto os argumentos fornecidos à chamada quanto os valores de retorno produzidos pela função alvo. No caso da função *NtQueryInformationProcess*, por exemplo, é possível identificar o identificador do processo alvo, a classe de informação requisitada *ProcessInformationClass* e o ponteiro de retorno de dados, elementos que permitem inferir se

⁵<https://github.com/contradef/Contradef/blob/main/Contradef/FunctionInterceptor.cpp>

```
[+] NtQueryInformationProcess...
Thread: 0
Call Id: 0
Routine address: 0x7ff8032ad800
Parameters:
  ProcessHandle: 0xfffffffffffff
  ProcessInformationClass: 7
  ProcessInformation: 0x86709cfc30
  ProcessInformationLength: 8
  ReturnLength: 0x0
Address of calling function: 0x7ff61469c679
[-] Start of call NtQueryInformationProcess
Returned from NtQueryInformationProcess (NTSTATUS): 0x0
Successful operation (STATUS_SUCCESS).
(Read 256 bytes from ProcessInformation, displaying in hex): 79 c6 69 14 f6 7f 0 0 e0 20 7f 87 0 0 0 0 11 0
70 86 0 0 0 0 0 0 0 0 0 0 0 31 35 f9 ef 9f 1 0 0 6 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 30 fc 9c 70 86 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[-] Call from NtQueryInformationProcess completed
[*] Completed
```

Figura 5.5: Captura de informações da função *NtQueryInformationProcess*.

a API está sendo utilizada para detectar depuradores. A interceptação dessas funções viabiliza a reconstrução do comportamento evasivo, fornecendo indícios sobre a tentativa de reconhecimento do ambiente. Essa abordagem contribui significativamente para o diagnóstico de amostras maliciosas que adotam verificações contextuais antes de executar o programa protegido.

5.3.2.2 Trace de Funções Externas - *TraceFcnCall*

O módulo *TraceFcnCall*⁶ registra, de forma contínua, a sequência cronológica de cada rotina invocada pelo binário analisado, identificando o endereço de destino, o módulo de origem (DLL), o nome simbólico da função e a *thread* responsável pela chamada (Figura 5.6).

O campo *forced* indica que a identificação simbólica original falhou no *framework* Pin, exigindo o emprego de heurísticas alternativas baseadas em funções nativas do *Windows* (*WinAPI*). Tal necessidade é recorrente em binários protegidos, cujos símbolos são intencionalmente ocultados para dificultar a análise de engenharia reversa.

Nesses cenários, a ferramenta executa uma inspeção manual da estrutura do executável em memória para recuperar o nome da função. O processo inicia-se com a obtenção do caminho do módulo associado ao endereço de instrução via função *GetMappedFileNameA* ou consulta a uma lista de módulos registrados, procedendo-se ao carregamento e à análise sintática (*parsing*) dos cabeçalhos do formato *Portable Executable* (PE).

Ao mapear o *IMAGE_DOS_HEADER* e o *IMAGE_NT_HEADERS*, o algoritmo localiza o diretório de exportação (*IMAGE_DIRECTORY_ENTRY_EXPORT*) para extrair os endereços virtuais relativos (*Relative Virtual Addresses* – RVAs). A lógica de extração percorre a tabela *AddressOfFunctions* e utiliza a tabela *AddressOfNameOrdinals* como ponte para associar cada endereço ao seu respectivo nome contido na tabela *AddressOfNames*. Nos casos em que uma função não possui um nome exportado explicitamente, o algoritmo a rotula preventivamente como "*Unnamed Function*", garantindo que a estrutura de endereçamento ainda seja contabilizada para o mapeamento do módulo.

⁶<https://github.com/contradef/Contradef/blob/main/Contradef/TraceFcnCall.cpp>

Uma vez que o formato PE não armazena o tamanho exato de cada função no diretório de exportação, o código implementa uma estratégia de reconstrução de intervalos. Os endereços coletados são ordenados de forma crescente, estabelecendo-se que o limite inicial (*start*) de uma rotina é o seu próprio RVA e o limite final (*end*) é o RVA da função imediatamente subsequente na memória. Essa segmentação cria uma série de intervalos contíguos, permitindo que a heurística resolva o nome da função mesmo quando o endereço capturado aponta para uma instrução intermediária no corpo da rotina, e não apenas para o seu ponto de entrada (*entry point*). Por fim, ao verificar em qual desses intervalos o endereço da chamada interceptada reside, a ferramenta recupera o símbolo correspondente, assegurando a continuidade do rastreamento mesmo sob condições de ofuscação severa.

Esse monitoramento fornece ao analista uma visão global do fluxo de execução, permitindo segmentar o comportamento do binário em fases distintas e identificar pontos críticos de interesse durante o processo de inicialização e execução.

Essa informação permite detectar padrões associados a comportamentos evasivos implementados por ferramentas de proteção, como o *Themida* e *Obsidium*. Esses protetores frequentemente empregam rotinas complexas que incluem chamadas artificiais à função `Sleep`, uso de APIs de verificação de ambiente (`IsDebuggerPresent`, `GetStartupInfoA`, `NtQueryInformationProcess`) e manipulação do fluxo de execução via tratamento estruturado de exceções (`RaiseException`, `RtlAddVectoredExceptionHandler`). A observação dessas chamadas, permite inferir estratégias de detecção de ambiente de execução voltadas à evasão de ferramentas de análise como depuradores, monitores de arquivos/registo e máquinas virtuais.

Para registrar as chamadas de forma efetiva, o módulo combina dois mecanismos complementares de interceptação. O primeiro mecanismo observa invocações diretas realizadas por instruções explícitas do tipo `call`, sendo eficaz para funções resolvidas pelo carregador do sistema operacional.

A Figura 5.6 apresenta um extrato típico do log produzido por esse primeiro mecanismo (M1), contendo informações estruturadas sobre as invocações capturadas. Cada linha do log exibe o endereço da função interceptada, o identificador da *thread*, o caminho do módulo carregado na memória e o nome da API monitorada. Na Figura 5.6 observamos funções como `GetCommandLineA`, `Sleep` e `RtlAddVectoredExceptionHandler`, que são comumente utilizadas por mecanismos de proteção para implementar técnicas de anti-análise, como atrasos programados, monitoramento de estado interno ou alteração dinâmica do fluxo de execução.

Endereço da função	Thread	Módulo, biblioteca ou imagem	Nome da função
1 7ffc36cc8c70	T[0]	C:\WINDOWS\System32\KERNEL32.DLL	GetCommandLineA
2 7ffc38374b80	T[0]	C:\WINDOWS\SYSTEM32\ntdll.dll	RtlAddVectoredExceptionHandler
3 7ffc38374b80	T[0]	C:\WINDOWS\SYSTEM32\ntdll.dll	RtlAddVectoredExceptionHandler
4 7ffc36cc8050	T[0]	C:\WINDOWS\System32\KERNEL32.DLL	RaiseException
5 7ffc38375c10	T[0]	C:\WINDOWS\SYSTEM32\ntdll.dll	RtlRemoveVectoredExceptionHandler
6 7ffc36cc8670	T[5]	C:\WINDOWS\System32\KERNEL32.DLL	Sleep
7 7ffc36546650	T[5]	[forced] \Device\HarddiskVolume3\Windows\System32\user32.dll	FindWindowA

Forced -> Indica que o Pin não reconheceu o nome da função, foi obtida usando outro método

Figura 5.6: Trecho de log produzido pelo módulo *TraceFncCall* (M1).

O segundo mecanismo (M2) apresentado na Figura 5.7 monitora saltos indiretos originados a partir de endereços obtidos dinamicamente — por exemplo, por meio das funções `GetProcAddress` e `LoadLibrary` — garantindo cobertura também em cenários de resolução tardia de símbolos ou técnicas de ofuscação de fluxo de controle. Essa abordagem combinada assegura robustez na análise, mesmo em ambientes hostis ou fortemente ofuscados que implementam diferentes estratégias para ofuscar ou mascarar a invocação de funções.

```

Call from 0x00007ff7c1470811 | UNKNOWN-IMG:main (rtn-id: 0) ->
Endereço função chamada 0x00007ff802bbae0 | Biblioteca USER32.dll | Função chamada FindWindowA (rtn-id: 11881) | Thread [T0] | Parâmetro 1 (endereço) 0x7ff7c1616dd1 -> | Valor parâmetro 1 Registry Monitor
Ret from 0x00007ff802bbaaf7 | USER32.dll:FindWindowA+0x00000000000017 (rtn-id: 11881) [T0] Return value ->
0 (next 0x7ff7c1929d48, came-from 0x7ff7c192323c)

```

Figura 5.7: Exemplo de log produzido pelo módulo *TraceFcnCall* M2

Mais do que enumerar chamadas, o encadeamento temporal desses eventos permite ao analista inferir a **semântica** do comportamento observado, ou seja, “o que” a amostra pretende fazer, e não apenas “como”. Por exemplo, um bloco de chamadas contendo `CryptAcquireContextW`, `CryptGenRandom`, `CryptImportKey` e, em seguida, ciclos de `FindFirstFileW`/`FindNextFileW`, `CreateFileW`, `ReadFile` e `WriteFile` revela um fluxo típico de *ransomware*: (i) inicialização do provedor criptográfico; (ii) geração ou importação de chaves; (iii) enumeração do sistema de arquivos; e (iv) regravação de dados cifrados. Essa abstração de alto nível complementa a granularidade dos demais módulos da *Contradef*, oferecendo suporte ao desenvolvimento de contramedidas específicas.

5.3.2.3 Rastreamento de Memória - *TraceMemory*

O módulo *TraceMemory*⁷ foi projetado para monitorar operações de leitura e escrita em regiões de memória acessadas durante a execução do binário analisado. Seu funcionamento é essencial para revelar informações que não estão presentes na imagem estática do executável, como dados decifrados em tempo de execução, *strings* sensíveis ou trechos de código desempacotados dinamicamente.

Durante a execução, o módulo registra, para cada acesso monitorado:

- o endereço de memória acessado;
- a instrução responsável pela operação;
- o valor lido ou escrito;
- e, quando aplicável, o conteúdo textual da memória acessada (ASCII ou Unicode).

Essa característica possibilita a extração de evidências como nomes de janelas, nomes de processos ou arquivos, chaves de registro e resoluções dinâmicas de APIs. Estes elementos frequentemente indicam mecanismos de evasão ou comportamentos maliciosos.

Essa capacidade de capturar o conteúdo de memória em pontos críticos da execução fornece ao analista uma visão privilegiada do comportamento interno do *malware*, complementando os demais módulos da ferramenta.

A Figura 5.8 apresenta um exemplo de rastreamento de escrita e leitura em memória contendo uma *string* associada à janela de um monitor de sistema, o que pode indicar tentativa de evasão de ferramentas como *Process Monitor*.

```

Thread: 0x00007ff7c1945d0c | Módulo: UNKNOWN-IMG | Função: main (rtn-id: 0) | Instrução (asm): mov dword ptr [rsi], edx | Operação: [Write] | Endereço de memória: 0x00007ff7c15ae380 | Valor (pilha): 0x434f5250 (1129271888) -> | Texto no ponteiro: "PROCmonclass"
[T3] 0x00007ff7c1945d0e | UNKNOWN-IMG:main (rtn-id: 0) | pop rdx | [Read 1 Op]: [0x00000083cb4ff850] = 0x0000000000000000 (0)

```

Figura 5.8: Extrato capturado pelo *TraceMemory* contendo dados da memória.

⁷<https://github.com/contradef/Contradef/blob/main/Contradef/TraceMemory.cpp>

Portanto, ao registrar o comportamento de leitura e escrita em memória com alto grau de detalhe, o módulo *TraceMemory* oferece subsídios concretos para a identificação de texto decifrado, relacionado à detecção de ambiente e outras técnicas de evasão, funcionando como uma ferramenta de apoio à análise comportamental dinâmica.

5.3.2.4 Rastreamento de Instruções - *TraceInstructions*

O módulo *TraceInstructions*⁸ realiza instrumentação em nível de instrução, registrando sequencialmente cada endereço executado, o conteúdo da instrução (em formato simbólico *assembly*), as *flags* e os operandos acessados ou modificados em tempo de execução.

Essa granularidade é útil para a análise de binários ofuscados ou protegidos, permitindo observar diretamente como o estado interno do processo é manipulado ao longo da execução. A coleta abrange instruções aritméticas, lógicas, de controle de fluxo (e.g., `call`, `jmp`, `ret`) entre outras.

A Listagem 5.3 apresenta um exemplo das instruções rastreadas pelo módulo, demonstrando operações de comparações, movimentação de registradores, carregamento de parâmetros a partir da pilha e outros resultados das operações das instruções, fornecendo uma observação detalhada do fluxo de execução, com ênfase na compreensão de lógicas internas.

Listagem 5.3: Extrato do *assembly* capturado no trace de instruções.

0x7ff77d513d48	hw_Obs.exe	mov rcx , qword ptr [rsp+0x70]	rcx = 0xffffffffffffffff
0x7ff77d513d4d	hw_Obs.exe	mov rdx , qword ptr [rsp+0x78]	rdx = 0x1f
0x7ff77d513d52	hw_Obs.exe	mov r8 , qword ptr [rsp+0x80]	r8 = 0xa5e01cf74c
0x7ff77d513d5a	hw_Obs.exe	mov r9 , qword ptr [rsp+0x88]	r9 = 0x4
...	...	...	
0x7ff77d51ffef	hw_Obs.exe	cmp dword ptr [rbp+0x18] , 0x4	[0xa5e01cf748] 0x4 == 0x04
0x7ff77d51fff3	hw_Obs.exe	jmp 0x7ff77d51fff7	
0x7ff77d51fff7	hw_Obs.exe	jnz 0x7ff77d520010	
0x7ff77d51fff9	hw_Obs.exe	jmp 0x7ff77d51fffd	
0x7ff77d51fffd	hw_Obs.exe	cmp dword ptr [rbp+0x1c] , 0x0	[0xa5e01cf74c] 0x0 == 0x00

Este rastreamento detalhado é especialmente valioso para:

- identificar transformações internas e transições de estado provocadas por instruções;
- recuperar dados armazenados temporariamente em registradores;
- detectar algoritmos de evasão ou maliciosos embutidos;
- localizar instruções artificiais utilizadas como armadilhas contra ferramentas de análise.

Em resumo, o módulo *TraceInstructions* oferece uma janela detalhada sobre a execução real de cada instrução no processo monitorado, desempenhando papel fundamental na detecção de técnicas evasivas e na engenharia reversa de *malwares*.

5.3.2.5 Desmontagem Dinâmica do Código Assembly - *TraceDisassembly*

O módulo *TraceDisassembly*⁹ realiza a desmontagem dinâmica do código executado, à medida que os blocos de instruções são descobertos pelo framework Intel Pin em tempo de execução. Cada bloco é registrado antes de sua execução real, gerando um relatório contínuo e sintático das instruções já traduzidas para *assembly* simbólico, com endereços e operandos resolvidos.

Esse processo ocorre de forma passiva e não intrusiva, sem modificar o comportamento original do binário analisado. Como não há coleta de valores em registradores ou memória,

⁸<https://github.com/contradef/Contradef/blob/main/Contradef/TraceInstructions.cpp>

⁹<https://github.com/contradef/Contradef/blob/main/Contradef/TraceDisassembly.cpp>

o foco está na estrutura sintática do código, o que permite observar a topologia de execução pretendida pelo binário, ainda que certas instruções nunca venham a ser efetivamente executadas.

A Listagem 5.4 apresenta o formato do log gerado, em que cada linha contém o endereço virtual da instrução, o mnemônico e os operandos correspondentes, evidenciando a forma simbólica das instruções descobertas durante a instrumentação. Esse registro é útil para identificar sequências sintáticas características de empacotadores, como cadeias de saltos indiretos, blocos de instruções inválidas, padrões repetitivos de NOPs e instruções sobre registradores especiais (e.g., RIP) com uso frequente em mecanismos de evasão.

Listagem 5.4: Extrato do *assembly* capturado no trace de desmontagem.

```
0x7ff7e3fb5087 | call qword ptr [rip+0x161d3]
0x7ff7e3fa9ac4 | sub rsp, 0x28
0x7ff7e3fa9ac8 | test rcx, rcx
0x7ff7e3fa9acb | jz 0x7ff7e3fa9ade
0x7ff7e3fa9acd | lea rax, ptr [rip+0x3435c]
0x7ff7e3fa9ad4 | cmp rcx, rax
0x7ff7e3fa9ad7 | jz 0x7ff7e3fa9ade
```

Além de auxiliar na identificação de padrões estruturais, o módulo permite localizar rotinas já desempacotadas, servindo como base para correspondência com dados dinâmicos extraídos por módulos complementares, como o *TraceInstructions* e o *TraceMemory*. Assim, o *TraceDisassembly* é uma escolha estratégica para análises complementares, quando o analista deseja mapear rapidamente a topologia do binário.

Em síntese, o módulo *TraceDisassembly* amplia a capacidade de análise estrutural da ferramenta *Contradef*, oferecendo uma visão do fluxo pretendido de execução, contribuindo para a detecção de artefatos sintáticos e comportamentos associados a técnicas de evasão e empacotamento.

5.3.2.6 Analisador de sequência de código executado - *InstructionSequenceDetector*

Para identificar e alterar as rotinas dinamicamente, em tempo de execução, aprimoramos a instrumentação binária dinâmica com a inclusão de um analisador de sequência de código executado que chamamos de **Detector de Sequências de Instruções (DSI)**, implementado no módulo *InstructionSequenceDetector*¹⁰. Este analisador utiliza uma fila que armazena várias instruções executadas pelo programa-alvo. A sequência de instruções previamente definida é comparada com a fila de execução. Quando há uma coincidência entre a sequência monitorada e as instruções executadas, um *callback* é disparado. Este *callback* pode ser configurado para executar procedimentos específicos antes da execução da última instrução da sequência. Essa abordagem permite controle sobre o fluxo de execução, possibilitando a execução de qualquer procedimento no *callback* antes da conclusão da sequência de instruções.

O **detector de sequencias de instruções** foi desenvolvido para detectar ocorrências de sequências específicas de instruções durante a execução de um programa. Seu objetivo principal é permitir a inspeção e coleta de dados em pontos específicos da execução, além da alteração no fluxo de controle, permitindo também a implementação de contramedidas para técnicas evasivas.

A implementação contém a definição de estruturas de dados que representam instruções instrumentadas e instruções a serem correspondidas. A estrutura *InstrumentedInstruction* armazena informações sobre cada instrução capturada durante a execução, incluindo seu endereço, a representação em string da instrução, uma versão normalizada dessa string (para facilitar

¹⁰<https://github.com/contradef/Contradef/blob/main/Contradef/InstructionSequenceDetector.cpp>

comparações) e informações sobre a classe da instrução (*OPCODE*). Já a estrutura *InstructionToMatch* representa as instruções que desejamos detectar, contendo campos semelhantes, mas com a capacidade adicional de lidar com curingas e instruções que podem corresponder a qualquer tipo (instruções anônimas).

A classe principal, *InstructionSequenceDetector*, gerencia o processo de detecção das sequências de instruções. Ela mantém uma lista de sequências alvo que foram adicionadas para monitoramento, além de um cache de instruções já processadas para otimizar o desempenho. A detecção é realizada por meio da comparação das instruções executadas pelo programa-alvo com as sequências armazenadas. Para isso, a classe utiliza uma fila que mantém as últimas instruções executadas, respeitando o tamanho máximo da maior sequência a ser detectada. A lógica do **detector de sequências de instruções** está representada na Figura 5.9.

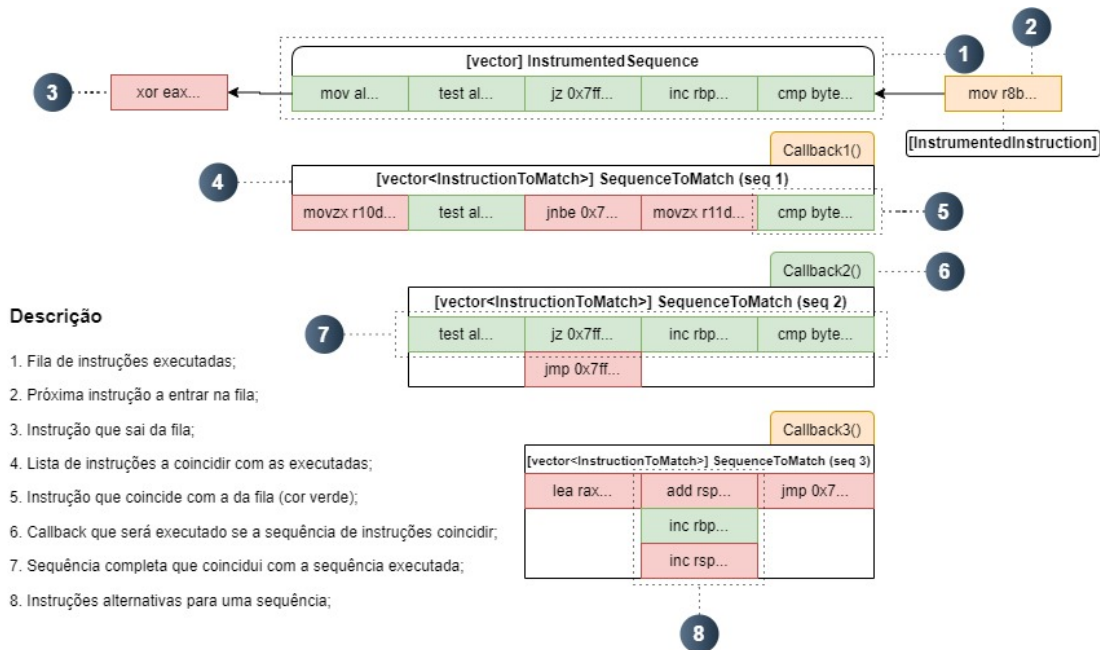


Figura 5.9: Fila de Instruções Instrumentadas e Sequências de Instruções a Serem Correspondidas

Durante a execução do programa instrumentado, cada instrução é interceptada e processada. A função de instrumentação insere um *callback* que é chamado antes da execução de instruções relevantes para as sequências de interesse. Esse *callback* atualiza a fila de instruções recentes e verifica se há correspondência com alguma das sequências alvo. A correspondência considera não apenas a igualdade exata das instruções, mas também permite o uso de curingas, possibilitando uma detecção mais flexível que pode, por exemplo, ignorar operandos específicos ou aceitar qualquer instrução em determinada posição da sequência.

Quando uma sequência é detectada, a classe verifica se há um *callback* associado a essa sequência e, caso positivo, executa o mesmo. Isso permite que ações específicas sejam realizadas quando padrões de instruções são encontrados, como modificar o comportamento do programa-alvo, coletar informações adicionais ou contornar técnicas evasivas. Essa capacidade é especialmente útil para alterar o comportamento do programa sob análise, principalmente os que empregam técnicas de ofuscação, criptografia e anti-depuração para impedir a análise

estática e dinâmica. A Listagem 5.5 exibe um exemplo de um *callback* que imprime o texto referenciado por um ponteiro armazenado no registrador RCX.

Listagem 5.5: Exemplo de um Callback para uma Sequência de Instruções

```
1 VOID SequenceMatchReadStringFcnCallback(THREADID tid, ADDRINT ip, CONTEXT* ctxt,
   SequenceInstructions& sequence)
2 {
3     ADDRINT regRcx = PIN_GetContextReg(ctxt, REG_RCX);
4
5     if (PIN_CheckReadAccess(reinterpret_cast<VOID*>(regRcx)))
6     {
7         std::cout << "RCX = 0x" << std::hex << regRcx << std::endl;
8         char* CharStr = reinterpret_cast<char*>(regRcx);
9         std::cout << "Nome da funcao: " << CharStr << std::endl;
10    }
11 }
```

A implementação também considera ambientes multi-thread. Dados específicos de thread são utilizados para garantir que a detecção de sequências seja precisa em cada contexto de execução, evitando interferências entre diferentes threads.

Uma parte importante da implementação é a normalização das instruções. Antes de serem comparadas, as instruções são processadas para remover espaços, converter para minúsculas e substituir endereços específicos por placeholders genéricos. Isso garante que instruções semanticamente equivalentes sejam reconhecidas como iguais, mesmo que apresentem variações superficiais.

Um ponto importante observado durante o uso do *detector de sequência de instruções* da **Contradef** é que sequências muito curtas ou excessivamente genéricas tendem a aumentar significativamente o risco de falsos positivos. Isso ocorre porque, no conjunto de instruções x86/x64, padrões de poucas instruções (por exemplo, combinações simples de `push/mov/call` ou blocos de preparação de *stack frame*) aparecem com alta frequência em código legítimo de aplicações Windows, bibliotecas de terceiros e até no código gerado automaticamente por compiladores. Quanto menor a sequência, maior a probabilidade de que ela seja apenas um fragmento comum de prólogo/epílogo ou de acesso a estruturas internas, e não um traço real da sequência de código desejada. Além disso, muitos protetores comerciais empregam ofuscação polimórfica, fazendo pequenas variações sintáticas em torno do mesmo padrão de evasão; caso o detector seja configurado com assinaturas muito genéricas, ele pode coincidir com essas variações e também com código benigno que compartilha a mesma forma superficial. Por esse motivo, a **Contradef** permite (i) definir um comprimento mínimo de sequência, (ii) associar a detecção a um contexto de execução (p. ex., “esta sequência apareceu logo após `IsDebuggerPresent`” ou dentro de um escopo identificado pelo módulo integrado ao *YARA*) e (iii) relacionar o achado com eventos de API ou memória. Esse encadeamento de evidências reduz a chance de marcar como evasivo um trecho que, isoladamente, é apenas comum.

5.3.2.7 Contramedidas para técnicas *anti-DBI* e *anti-VM*

Implementamos na *Contradef* um conjunto de contramedidas inspiradas nas descritas por Lee et al. (2021) para os protetores comerciais *Obsidium* e *Themida*, adaptadas e estendidas para a arquitetura x64. Em linhas gerais, nossas defesas combinam: (i) interceptação de APIs sensíveis (p. ex., consultas a firmware, registro e módulos do sistema), (ii) reescrita de *buffers* de saída e (iii) sanitização temporária de *strings* indicativas de VM/monitoramento antes da chegada da resposta ao código protegido.

O estudo de Lee et al. (2021) fornece uma visão de alto nível das famílias de técnicas *anti-VM* (p. ex., leitura de tabelas de firmware, inspeções de *registry*, verificação de processos/serviços de virtualização) e *anti-DBI* (p. ex., sondagens de API e exceções para sinalizar instrumentação)

empregadas por protetores comerciais. A Tabela 5.1 resume, conforme apresentado por Lee et al. (2021), os vetores mais característicos observados por protetor. Essa sistematização nos guiou na priorização e desenho das contramedidas, que implementamos de forma independente e transparente ao binário alvo, buscando compatibilidade com variações de versões.

Tabela 5.1: Resumo das técnicas *anti-VM* e *anti-DBI* utilizadas em protetores comerciais (síntese de Lee et al., 2021)

Protector	Anti-VM Techniques	Anti-DBI Techniques
Themida	- Registry - Hardware	- N/A
Obsidium	- Hardware (IN Instruction) - Hardware (Disk Drive Value, Firmware Table Information) - Process (VBoxGuest)	- API ZwQueryInformationProcess (0x1f) - Process (VBoxGuest) - Exception (Single Step)

No contexto deste trabalho, empregamos (e avaliamos) contramedidas alinhadas aos *princípios* mapeados por Lee et al. (2021), mas com implementação própria e compatível com **x64**. Em particular, nossas estratégias incluem: mascaramento de artefatos de VM em consultas a firmware/ACPI e registro; normalização de listas de módulos do sistema; e limpeza de *strings* sensíveis em *buffers* retornados por funções APIs, reduzindo a eficácia de verificações *anti-VM* e *anti-DBI* sem alterar a lógica da amostra protegida. Por consequência, mitigamos a superfície de detecção e também o volume de *logs* gerados por iterações redundantes, favorecendo a reprodutibilidade dos experimentos.

5.4 Limitações

Apesar dos benefícios apresentados, a *Contradef* possui algumas restrições práticas:

1. **Suporte restrito a executáveis PE 64-bit:** a versão atual da ferramenta é compatível apenas com binários nativos de 64 bits no formato *Portable Executable*. Binários PE 32-bit são parcialmente suportados e artefatos desenvolvidos em *.NET*, *Java*, linguagens interpretadas (como PowerShell, JavaScript ou VBA) não são suportados, o que limita a aplicação da ferramenta a um subconjunto específico dos *malwares* que circulam em ambientes Windows.
2. **Dependência de operações de entrada/saída em disco:** a geração contínua de registros detalhados pode introduzir gargalos de desempenho, especialmente em sistemas equipados com discos magnéticos. Unidades *SSD* tendem a mitigar esse impacto, sobretudo nos modos que produzem grandes volumes de saída, como *TraceMemory* e *TraceInstructions*.
3. **Geração de tamanhos excessivos dos arquivos de rastreamento:** binários que realizam processamento intensivo ou permanecem em execução prolongada podem gerar arquivos de *trace* com dezenas de gigabytes, exigindo considerável espaço em disco e aumentando o custo computacional do pós-processamento.
4. **Sensibilidade a técnicas de evasão anti-DBI capazes de detectar aumento de overhead:** alguns mecanismos descritos na literatura Polino et al. (2017a); Gozzini et al.

(2022) são capazes de detectar a sobrecarga introduzida pela instrumentação dinâmica. Modos com maior granularidade, como *TraceMemory* e *TraceInstructions*, tendem a intensificar esse *overhead*, tornando-se mais suscetíveis à detecção.

5. **Necessidade de interrupção manual em execuções prolongadas:** programas que não encerram sua execução de forma autônoma exigem finalização manual por parte do analista, após a coleta dos dados de interesse. Caso contrário, a ferramenta continuará a registrar eventos indefinidamente, comprometendo o uso eficiente de recursos.
6. **Variabilidade no tempo de execução, de acordo com o perfil de instruções do binário analisado:** aplicações que envolvem muitos saltos indiretos, chamadas frequentes a APIs ou rotinas criptográficas executadas em *loop* tendem a prolongar o tempo de coleta e a aumentar o consumo de espaço em disco.

5.5 Discussão

A ferramenta *ContraDef* foi projetada para análise de malware assim como de comportamentos evasivos adotados por empacotadores comerciais, como *Obsidium* e *Themida*, quando empregados na proteção de executáveis potencialmente maliciosos.

A *ContraDef* também adota um modelo de pré-análise estática utilizando regras *YARA* que são procuradas diretamente no binário instrumentado. Essa funcionalidade é eficaz na identificação antecipada de padrões estruturais específicos de cada protetor, permitindo a configuração automatizada da instrumentação com base no protetor detectado. Com isso, evita-se a sobreinstrumentação e melhora-se a assertividade da coleta de dados. A identificação prévia do protetor utilizado possibilita uma delimitação mais precisa do escopo de instrumentação, permitindo que estratégias específicas sejam ativadas dinamicamente de acordo com o protetor identificado. Otimizando o tempo e os recursos computacionais investidos durante a análise. Essa flexibilidade contribui para tornar a análise mais direcionada e eficiente, focando em técnicas evasivas utilizadas pelas ferramentas de proteção *Obsidium* e *Themida*.

A arquitetura modular da *ContraDef* permite que módulos específicos — como *TraceFcn-Call*, *TraceMemory* e *TraceInstructions* — sejam ativados de acordo com as necessidades do analista, controlando o volume de dados coletados e o tempo de execução. A possibilidade de relacionar os logs coletados pelos módulos em múltiplos níveis de abstração contribui para uma compreensão mais abrangente e precisa do comportamento executado. Ao integrar múltiplas perspectivas — como rastreamento de chamadas, instruções e acesso à memória — torna-se possível inferir padrões complexos associados a mecanismos de evasão. Por exemplo, cadeias de caracteres extraídas da memória podem ser relacionadas a instruções específicas e ao contexto de execução em que foram manipuladas, permitindo inferir sua origem, finalidade e momento de ativação. Da mesma forma, padrões identificados no rastreamento de chamadas podem revelar interações sistemáticas com o ambiente — como consultas a informações de processo, hardware ou configuração do sistema — frequentemente utilizadas por empacotadores para detectar depuradores ou máquinas virtuais.

Esse encadeamento de evidências, extraídas de múltiplos pontos de observação, amplia a capacidade interpretativa da análise dinâmica, permitindo ao analista reconstruir o fluxo lógico do binário e identificar comportamentos suspeitos de forma mais robusta, mesmo diante de técnicas avançadas de ofuscação e evasão.

Em particular, o **Detector de Sequências de Instruções (DSI)** mostrou-se um componente flexível e promissor para a detecção e manipulação de padrões de código associados a mecanismos de evasão e comportamento malicioso. Além de identificar tais padrões, ele permite

ao analista interagir ativamente com a execução em nível de instrução—alterando o comportamento do programa em pontos críticos e implementando contramedidas personalizadas—o que viabiliza a observação de comportamentos complexos e a detecção de técnicas de anti-análise. Sua implementação considera desafios práticos, como eficiência, precisão em ambientes multithread e a necessidade de contornar mecanismos de proteção, mantendo transparência suficiente para sustentar a coleta de evidências; assim, a abordagem deixa de ser meramente passiva e passa a ser ativa, permitindo ações corretivas durante o próprio processo de instrumentação.

Embora a **Contradef** tenha sido concebida com foco em programas evasivos, seus recursos aplicam-se igualmente à inspeção de binários legítimos sempre que se deseja observar, com alta granularidade, o comportamento interno de um executável; entre suas principais virtudes destacam-se: (i) o **Detector de Sequências de Instruções**, que permite redirecionar saltos e manipular dinamicamente o comportamento do programa alvo; (ii) o uso de regras *YARA* para concentrar a instrumentação em pontos relevantes, reduzindo sobrecarga e volume de *logs*; (iii) módulos para a coleta de rastreamento de memória e instruções, que registram eventos de interesse (por exemplo, chamadas de API ou dados críticos em memória); e (iv) a capacidade de registrar e manipular parâmetros e valores de retorno, forçando a execução de caminhos que o *malware* só percorreria sob condições específicas. Este conjunto de características amplia a eficácia da análise e possibilitam a execução bem sucedida de software protegido mesmo em ambientes virtuais.

Capítulo 6

Experimentações e Análises

Este capítulo apresenta a execução prática da ferramenta *Contradef* em programas de testes protegidos por diferentes versões dos protetores comerciais *Themida* e *Obfuscator*, com o objetivo de validar sua eficácia na instrumentação e coleta de dados para análise. Os experimentos foram conduzidos com foco na verificação do funcionamento correto dos módulos implementados, na análise da transparência da instrumentação (isto é, a ausência de efeitos colaterais perceptíveis durante a execução) e na avaliação do impacto em termos de desempenho (overhead).

A diversidade de versões utilizadas visa demonstrar a adaptabilidade da ferramenta frente às variações introduzidas por atualizações ou modificações dos protetores, que podem influenciar diretamente a dinâmica do código protegido. Ao longo dos testes, foram coletadas evidências comportamentais, comparados tempos de execução e analisada a integridade da instrumentação, permitindo uma avaliação crítica sobre a robustez e os limites atuais da solução proposta.

6.1 Prova de Conceito

A prova de conceito foi conduzida por meio da execução da ferramenta *Contradef* em amostras de teste benignas protegidas por diferentes versões dos protetores *Themida* e *Obfuscator*. O principal objetivo dos experimentos foi avaliar a eficácia da instrumentação em termos de geração bem-sucedida dos logs e rastreamentos (*traces*), o impacto no tempo de execução (overhead) e o grau de transparência da ferramenta frente à execução de programas protegidos.

Além disso, buscou-se verificar se a presença da instrumentação interferia no comportamento esperado das amostras protegidas, garantindo que as análises conduzidas refletissem a lógica real de execução, sem detecções ou desvios causados pela presença do instrumentador.

Durante os testes, também foram avaliados o volume dos dados gerados por cada módulo e a estabilidade da ferramenta diante de diferentes técnicas de proteção empregadas pelos protetores.

Os experimentos foram conduzidos em ambiente virtualizado, a fim de simular condições realistas de análise de *malware* — comumente realizadas em máquinas virtuais — e garantir a reprodutibilidade dos resultados. Para mensurar o impacto individual, os principais módulos da *Contradef* (`FunctionInterceptor`, `TraceFncCall`, `TraceDisassembly`, `TraceMemory` e `TraceInstructions`) foram executados de forma isolada. Já o módulo `InstructionSequenceDetector` permanece habilitado por padrão e é executado em conjunto com qualquer módulo selecionado ou quando nenhum módulo é ativado, pois aciona contramedidas genéricas e específicas e opera acoplado à instrumentação básica.

Além das execuções isoladas, considerou-se também o cenário com **todos os módulos** habilitados simultaneamente, a fim de observar o comportamento agregado do sistema sob instrumentação intensiva. Adicionalmente, realizou-se a execução *sem* a *ContradeF*, utilizada como referência (*baseline*) para comparação.

Para aferir uma linha de base de desempenho para o programa em sua versão original, isto é, sem a instrumentação da *ContradeF*, foi necessário executar o binário em um ambiente nativo (*bare metal*). Essa abordagem foi adotada porque o executável, na ausência das contramedidas, possui mecanismos que detectam a execução em ambientes virtualizados, resultando em seu encerramento prematuro. Consequentemente, a medição do tempo de execução em uma máquina virtual (*VM*) não constituiria uma linha de base confiável, uma vez que o programa não completaria suas rotinas relevantes.

Contudo, é imperativo reconhecer que uma execução nativa apresenta, intrinsecamente, um desempenho superior ao de um ambiente virtualizado. Tal disparidade decorre de dois fatores principais: (i) a ausência da camada de abstração do hipervisor, eliminando os custos computacionais associados à virtualização de instruções e à emulação de dispositivos; (ii) o acesso direto e irrestrito aos recursos de *hardware* do hospedeiro (CPU, memória, I/O), que otimiza a performance geral. Portanto, a inerente redução no tempo de execução observada no ambiente *bare metal* é uma variável que deve ser considerada ao comparar estes resultados com as medições obtidas sob virtualização. Assim, mesmo a medição em *bare metal*, embora necessária para evitar a detecção, não representa uma linha base confiável para comparação com os cenários instrumentados em VM devido às diferenças intrínsecas de desempenho entre os ambientes. Portanto, os valores reportados devem ser interpretados como *estimativas do overhead* introduzido pelos modos de instrumentação, e não como equivalências diretas entre a execução nativa e a virtualizada.

Ademais, as versões dos protetores avaliadas exibem uma janela modal inicial; para não contaminar as medições, essa janela foi fechada manualmente imediatamente após sua aparição, procedimento repetido de forma consistente em todas as execuções.

6.1.1 Ambiente de Experimentação

Máquina hospedeira. A máquina hospedeira utilizada possui um processador Intel Core i7-1165G7, 16 GB de memória RAM e uma unidade SSD SATA III (Crucial CT1000BX500SSD1) de 1 TB executando no sistema operacional Windows 11 x64 (versão 24H2). O *hypervisor* adotado foi o Oracle VirtualBox versão 7.0.

Máquina virtual. O sistema operacional convidado é o Windows 10 x64 (versão 22H2), configurado com 5 GB de memória RAM e seis VCPUs mapeadas diretamente para os núcleos físicos do processador. O disco da VM possui alocação fixa, com 200 GB de espaço reservado.

Condições de execução. A ferramenta *ContradeF* foi executada diretamente por meio do binário `pin.exe`, em diferentes configurações: execução individual de cada módulo de instrumentação, execução combinada de todos os módulos simultaneamente e execução sem a *ContradeF* (em *bare-metal*).

Métricas de desempenho. O tempo total de execução para cada cenário foi medido utilizando o comando `Measure-Command` do PowerShell, fornecendo uma estimativa do overhead introduzido pela instrumentação. Além disso, o tamanho final dos arquivos de *log* gerados em cada configuração foi registrado, permitindo avaliar o impacto da análise em termos de consumo de armazenamento.

6.1.2 Resultados da Execução com Amostras de Teste

Esta seção apresenta os resultados obtidos com a aplicação da ferramenta *Contradef* sobre amostras de teste benignas. Para este propósito, foi desenvolvido um programa mínimo de avaliação que imprime apenas a mensagem "Hello World" no console, garantindo que a carga computacional intrínseca ao programa seja praticamente nula.

Esse programa foi protegido com diferentes versões dos protetores *Obsidium* e *Themida*, permitindo que a análise se concentrasse exclusivamente no impacto introduzido pelos mecanismos de proteção. Dessa forma, o comportamento observado durante a execução reflete majoritariamente o funcionamento dos protetores, e não do binário em si.

O objetivo central desses experimentos foi avaliar o impacto da execução da *Contradef* sobre as variantes protegidas do programa de teste, considerando: (i) o tempo total de execução; (ii) o volume dos resultados gerados (arquivos de *log* e *traces*); e (iii) a transparência frente aos mecanismos anti-análise dos protetores, como técnicas de detecção de ambientes virtuais (anti-VM) e de instrumentação binária dinâmica (anti-DBI).

Os dados coletados nos experimentos foram organizados em tabelas e analisados sob diferentes perspectivas, com o objetivo de avaliar o impacto da *Contradef* na execução de binários protegidos.

A análise considerou métricas quantitativas (tempo de execução e volume de saída de *logs*) e qualitativas (transparência da ferramenta frente a técnicas de anti-análise). Os cenários instrumentados foram comparados a uma linha de base obtida em execução nativa, isto é, sem instrumentação.

A execução bem-sucedida do programa de teste protegido constitui um indicador direto da integridade do processo de análise. Quando o texto "Hello World" é exibido no console, infere-se que o binário foi corretamente descompactado e executado, sem interferência crítica por parte dos mecanismos de proteção ou da instrumentação aplicada. Essa característica é especialmente relevante no contexto de validação da ferramenta *Contradef*, uma vez que, em amostras reais de *malware*, torna-se significativamente mais difícil determinar se a execução ocorreu conforme o esperado.

Nesse sentido, o uso de um binário minimamente funcional como amostra de teste fornece um ponto de referência confiável para avaliar tanto o impacto da instrumentação quanto a transparência da execução. Adicionalmente, a simplicidade estrutural do programa permite isolar, com maior precisão, os efeitos provocados exclusivamente pela atuação do protetor ou pelos módulos da ferramenta. Como a carga computacional intrínseca ao binário protegido é deliberadamente reduzida, qualquer desvio de comportamento, atraso ou falha durante a execução pode ser associado, com maior grau de confiança, à interação entre os mecanismos de proteção e os módulos de análise da *Contradef*.

Tal abordagem viabiliza estimativas mais precisas sobre o *overhead* introduzido pelos diferentes modos de instrumentação, além de permitir a verificação empírica da eficácia das contramedidas implementadas contra técnicas de evasão, tais como anti-VM e anti-DBI. O cenário experimental delineado oferece uma base controlada, reproduzível e tecnicamente justificável para validar o comportamento da ferramenta em condições adversas, simulando, com fidelidade, as situações enfrentadas em ambientes reais de análise de binários protegidos. Desta forma, os dados coletados possibilitaram medir não apenas o *overhead* introduzido pela instrumentação, mas também a eficácia da *Contradef* em contornar mecanismos evasivos dos protetores testados por meio das contramedidas aplicadas.

Os resultados apresentados a seguir estão segmentados por protetor utilizado, e permitem avaliar o *overhead* e a transparência da ferramenta.

6.1.3 Execução do *Obsidium*

Foram avaliadas amostras de um programa de teste protegidas por três versões do protetor *Obsidium Demo* (1.4.5 *build* 31, 1.7.3 *build* 5 e 1.8.5 *build* 4), com o objetivo de mensurar a transparência da instrumentação frente às técnicas evasivas introduzidas por diferentes variações do protetor.

Os resultados apresentados nesta subseção referem-se à versão mais recente (*Obsidium 1.8.5 build 4*), uma vez que os comportamentos observados nas versões anteriores foram consistentes quanto a fluxo de execução, volume de *logs* e métricas temporais. Essa decisão evita redundâncias e preserva a representatividade para análise e discussão.

A Tabela 6.1 resume as principais métricas coletadas durante a execução da amostra protegida com *Obsidium 1.8.5 build 4*. As análises consideraram o tempo total de execução sob diferentes configurações da *Contradef*, o tamanho dos arquivos de *log* gerados por cada módulo e a eficácia das contramedidas diante de técnicas anti-instrumentação e anti-*VM* aplicadas pelo protetor.

Tabela 6.1: Métricas de execução nos diferentes cenários experimentais da versão 1.8.5 *build* 4.

Módulo ativado	Tempo de Exec.	Tamanho do Log	Exec. Bem-Sucedida
Sem Contradef (bare-metal)	741 ms	-	Sim
InstructionSequenceDetector	52 s 214 ms	-	Sim
FunctionInterceptor	1 min 12 s 288 ms	50 KB	Sim
TraceFcnCall	1 min 29 s 990 ms	534 KB	Sim
TraceDisassembly	1 min 14 s 221 ms	0.98 MB	Sim
TraceMemory	1 min 58 s 106 ms	1.01 GB	Sim
TraceInstructions	5 min 6 s 6 ms	13.9 GB	Sim
Todos	6 min 3 s 748 ms	15.0 GB	Sim

6.1.4 Execução do *Themida*

De forma análoga, foram avaliadas as versões 3.0.4, 3.1.3 e 3.1.8 do *Themida*, com o objetivo de mensurar o impacto da instrumentação em diferentes gerações do protetor e verificar a capacidade da *Contradef* de lidar com as técnicas evasivas. Assim como no caso anterior, analisaram-se o tempo de execução, o volume de dados gerados pelos módulos e a transparência da execução frente a mecanismos de anti-instrumentação e anti-*VM*.

Os resultados relatados a seguir concentram-se na versão *Themida 3.1.8*, dado que as versões anteriores apresentaram comportamento consistente em termos de fluxo de execução, *overhead* e geração de *logs*.

Tabela 6.2: Métricas de execução nos diferentes cenários experimentais da versão *Themida 3.1.8*.

Módulo ativado	Tempo de Exec.	Tamanho do Log	Exec. Bem-Sucedida
Sem Contradef (bare-metal)	480 ms	-	Sim
InstructionSequenceDetector	33 s 370 ms	-	Sim
FunctionInterceptor	1 min 12 s 39 ms	186 KB	Sim
TraceFcnCall	1 min 31 s 931 ms	28.7 MB	Sim
TraceDisassembly	1 min 10 s 879 ms	8.70 MB	Sim
TraceMemory	15 min 50 s 906 ms	17.8 GB	Sim
TraceInstructions	29 min 41 s 436 ms	49.4 GB	Sim
Todos	1 h 12 min 45 s 267 ms	70.1 GB	Sim

As Tabelas 6.1 e 6.2 sintetizam o impacto de cada módulo no tempo de execução e no volume de *logs*. Em termos gerais, os módulos de granularidade por instrução (`TraceInstructions` e `TraceMemory`) concentram o maior custo, ao passo que módulos que fazem intercepções pontuais (`FunctionInterceptor`, `TraceFcnCall` e `TraceDisassembly`) impõem sobrecarga reduzida.

No caso do *Themida*, o alto *overhead* decorre, em grande medida, da aplicação de *virtualização de código* por meio de uma *máquina virtual* (VM) como principal técnica de ofuscação. Em termos breves, a VM, no contexto deste protetor, é uma técnica que traduz instruções originais em bytecode virtual usando uma arquitetura de conjunto de instruções (ISA) personalizada e incorpora um interpretador de máquina virtual no programa alvo (Zhang et al., 2025).

Cada sistema de VM disponível no *Themida* implementa um método de proteção que pode gerar instâncias distintas, com variações em registradores e rotinas nativas (blocos de código x86/x64) que implementam a semântica de cada *OPCODE* por meio de tabelas de *OPCODEs* equivalentes, ampliando a diversidade estrutural entre os binários protegidos (Oreans, 2025).

Cada compilação (*build*) de programa protegido pelo *Themida* contém uma instância única dessa arquitetura, com variações no *layout* dos registradores, no mapeamento da tabela de *OPCODEs* e nas implementações das rotinas associadas aos *OPCODEs* (chamadas de *handlers*) (Oreans, 2025). Consequentemente, as implementações dessas rotinas mudam entre instâncias, podendo apresentar endereços, formas, convenções internas, sequências de instruções e combinações de operações distintas, como a inserção de código irrelevante (ex. sequências de saltos), além do rearranjo de registradores virtuais.

Para a instrumentação, essa mecânica de ofuscação se manifesta com um alto volume de saltos indiretos, invocações de rotinas associadas aos *OPCODEs* e acessos adicionais à memória (ex., para estados internos e tabelas de *OPCODEs*), de modo que uma única instrução original pode se expandir em centenas ou até milhares de instruções ofuscadas, tornando a engenharia reversa extremamente difícil (Zhang et al., 2025). Isso aumenta o número de instruções observadas e a densidade de eventos instrumentáveis, elevando o tempo sob instrumentação e o volume de *logs* — efeito especialmente perceptível nos modos que dispõem *callbacks* por instrução.

O *overhead* mensurado para cada módulo inclui a sobrecarga do `InstructionSequenceDetector`, que opera continuamente. Esse módulo não produz um arquivo de saída próprio, sendo, no entanto, necessário para a detecção de sequências de instruções relacionadas à aplicação de algumas contramedidas. A sobrecarga desse componente é, portanto, incorporada às métricas de todos os módulos analisados.

A diferença entre os tempos de `TraceInstructions` e `TraceMemory` com relação aos outros decorre principalmente da granularidade dos *hooks*: nesses módulos, os *callbacks* são disparados com alta frequência, enquanto, nos demais, os *hooks* são pontuais e associados a trechos específicos de código.

Os resultados confirmam que a sobrecarga cresce com a granularidade da instrumentação e que a virtualização de código empregada em protetores como o *Themida* intensifica o número de eventos observáveis e o volume de *logs*. Ainda que os módulos de intercepção pontual imponham custos reduzidos, a presença do componente `InstructionSequenceDetector` implica um custo adicional constante em cada execução. As tabelas apresentadas oferecem, portanto, um panorama quantitativo e qualitativo do comportamento do sistema sob diferentes configurações, sustentando as discussões subsequentes sobre a utilidade da ferramenta e implicações práticas para a análise dinâmica.

6.1.5 Limitações práticas e recomendações operacionais

- **Sobrecarga e E/S.** A instrumentação introduz *overhead* inevitável e gera grande volume de dados, especialmente em `TraceMemory` e `TraceInstructions`. Em *hardware* com discos magnéticos, a escrita contínua pode se tornar gargalo; recomenda-se *SSD* para mitigar o impacto de E/S.
- **Tamanho dos *traces*.** Amostras com execução prolongada, rotinas criptográficas em *loop*, uso intensivo de *APIs* ou alto nível de ofuscação podem produzir dezenas de gigabytes de *trace*, elevando custos de armazenamento e pós-processamento.
- **Detectabilidade do *overhead*.** Alguns mecanismos descritos na literatura [Polino et al. \(2017a\)](#); [Gozzini et al. \(2022\)](#) conseguem inferir a presença de instrumentação dinâmica. Modos com maior granularidade são mais suscetíveis a tal detecção.
- **Término da execução.** Programas que não finalizam autonomamente exigem encerramento manual após a coleta dos dados de interesse; do contrário, o registro de eventos prossegue indefinidamente.
- **Generalização limitada.** A análise concentrou-se em subconjuntos específicos de versões de *Obsidium* e *Themida*. Versões mais recentes ou variantes podem demandar ajustes de módulos e lógica de detecção.
- **Contramedidas para o *VirtualBox*.** As contramedidas anti-*VM* foram desenvolvidas e validadas especificamente para o *VirtualBox*; sua eficácia pode não se reproduzir em outros hipervisores (por exemplo, VMware ou Hyper-V), demandando adaptações adicionais.

As limitações identificadas não inviabilizam o uso da *ContraDef*, mas recomendam precauções operacionais: seleção criteriosa dos modos de instrumentação, provisionamento de armazenamento compatível com o volume esperado e, quando possível, execução em ambientes com alta capacidade de E/S de dados. Além disso, a generalização para outros hipervisores requer implementação adicional das contramedidas anti-*VM*.

6.1.6 Discussão

Os experimentos realizados mostraram que a ferramenta *ContraDef* conseguiu executar de forma satisfatória em diferentes cenários, abrangendo múltiplas versões de protetores utilizados em programas maliciosos. Todos os módulos funcionaram corretamente nas versões 1.4.5 *build* 31, 1.7.3 *build* 5, 1.8.1 *build* 4 e 1.8.5 *build* 4 do *Obsidium*, bem como nas versões 3.0.4, 3.1.3 e 3.1.8 do *Themida*.

Contudo, alguns pontos limitantes também foram observados. O uso de instrumentação binária dinâmica introduz inevitavelmente certo *overhead* na execução, o que pode afetar o desempenho e a escalabilidade da análise em amostras maiores ou em execuções prolongadas.

Outro aspecto a ser considerado é que as contramedidas aplicadas contra técnicas de detecção de máquinas virtuais foram especificamente desenvolvidas e validadas para o *VirtualBox*, podendo não apresentar a mesma eficácia em outros programas de virtualização, como VMware ou Hyper-V. Essa limitação sugere a necessidade de adaptações adicionais, caso a ferramenta seja empregada em ambientes heterogêneos.

Por fim, a análise concentrou-se em um conjunto limitado de versões dos protetores *Obsidium* e *Themida*. Embora os resultados sejam promissores, versões mais recentes ou configurações personalizadas nos protetores possam introduzir alterações significativas, exigindo outras contramedidas, ajustes nos módulos de rastreamento ou na lógica de detecção.

Os resultados validam a *Contradef* como uma ferramenta prática de análise dinâmica — modular e adaptável — capaz de fornecer indícios para a caracterização de técnicas evasivas e comportamentos maliciosos em executáveis protegidos por empacotadores comerciais; sua arquitetura orientada a módulos permite instrumentação direcionada e correspondência de múltiplos vetores de observação, ampliando significativamente o poder de caracterização. Ainda assim, persistem limitações relacionadas a desempenho, cobertura e necessidade de evolução contínua, de modo que a ferramenta pode ser empregada como parte de um ecossistema mais amplo de técnicas de análise, e não como solução isolada.

Ao mesmo tempo, os desafios enfrentados durante os experimentos ressaltam a complexidade dos empacotadores modernos e a constante evolução dos mecanismos de proteção, exigindo manutenção contínua e adaptação da ferramenta. Tais limitações evidenciam que, embora poderosa, a *Contradef* não deve ser encarada como solução única, mas sim como um componente complementar dentro de um ecossistema mais amplo de análise, que pode incluir ferramentas de sandboxing, máquinas virtuais e análise híbrida estática/dinâmica.

Capítulo 7

Análise com Amostras de Malwares

Este capítulo apresenta e discute um conjunto de técnicas evasivas empregadas pelas ferramentas de proteção de software *Obsidium* e *Themida*. A análise foi conduzida com o auxílio da *Contradef*, que permitiu identificar padrões de comportamento característicos de cada uma dessas soluções de empacotamento e ofuscação.

Ao longo da seção, são descritos os principais mecanismos utilizados por essas ferramentas para dificultar a análise e inspeção de executáveis, bem como a forma como tais mecanismos impactam a análise dinâmica e estática de *malwares*. A *Contradef* desempenhou um papel central nesse processo, pois possibilitou a detecção de rotinas específicas introduzidas por esses protetores, viabilizando a criação de contramedidas eficazes e a identificação das técnicas evasivas.

Com isso, este capítulo não apenas evidencia a sofisticação das técnicas evasivas implementadas por *Obsidium* e *Themida*, mas também demonstra como a instrumentação binária dinâmica pode ser aplicada de maneira eficaz para compreender, classificar e mitigar os obstáculos impostos por essas ferramentas.

7.0.1 Ambiente de Experimentação

Esta seção descreve o protocolo experimental adotado para avaliar a eficácia da ferramenta *Contradef* na identificação de técnicas evasivas em programas maliciosos. O experimento foi conduzido sobre duas amostras representativas de *malware*, selecionadas de acordo com critérios técnicos que asseguram a presença de mecanismos de autoproteção e evasão. O objetivo é verificar a capacidade dos módulos de rastreamento em fornecer indícios para caracterizar comportamentos evasivos em amostras reais de *malwares*.

7.0.1.1 Seleção das Amostras

Avaliamos a **Contradef** em amostras reais de *malware*, com o objetivo de verificar sua capacidade de revelar comportamentos evasivos em condições práticas de análise. As amostras foram selecionadas com base em correspondências detectadas pelas regras *YARA* desenvolvidas neste trabalho, que indicam alvos potenciais para estudo.

Para sistematizar esse processo, implementamos uma automação que realiza uma análise preliminar com a ferramenta *YARA*, utilizando as regras criadas para este estudo. Essa automação permitiu a triagem eficiente de um conjunto de 23.489 amostras provenientes do repositório MalwareBazaar¹, catalogadas entre 2022 e 2025.

Após a triagem inicial, as amostras que apresentaram correspondência com os padrões definidos foram selecionadas de forma arbitrária para análise detalhada, permitindo testar a ferramenta em cenários realistas e avaliar sua eficácia na detecção de técnicas evasivas.

As amostras selecionadas foram as seguintes:

- *Obsidium* - 0d0617fc515cd65ce750969d86f35563645173137d55b740cfd62163bc00cd6b
- *Themida* - a1c687574b512e60c254447937836a11aca2ef11a928cf28c5e9e9138f7ce934

7.0.1.2 Máquina virtual

O sistema convidado executa Windows 10 x64 (22H2), configurado com 5 GB de memória e seis VCPUs mapeadas diretamente para o processador físico. O armazenamento da VM é fixo, com 200 GB de espaço em disco. Antes da execução de cada amostra, todos os *snapshots* foram restaurados para um ponto limpo.

7.0.1.3 Condições de execução

A fim de mitigar possíveis interferências no ambiente de execução, recursos como o *Windows Defender* e o *User Account Control* (UAC) foram desabilitados. Adicionalmente, o terminal foi executado com privilégios administrativos para assegurar que os binários tivessem acesso irrestrito às chamadas de sistema, permitindo assim a exploração completa das funcionalidades das ferramentas de proteção.

7.0.2 Execução Monitorada

Durante a execução em ambiente controlado, a *ContradeF* foi executada em cada amostra, registrando eventos relevantes para a reconstrução do comportamento interno. Entre os dados coletados destacam-se:

- Instruções executadas em tempo real;
- Chamadas de função interceptadas;
- Acessos à memória monitorados;
- Trechos de código reconstruídos dinamicamente, incluindo rotinas geradas após desempacotamento.

A análise desses registros permitiu relacionar eventos instrumentados com os comportamentos observados, possibilitando a reconstrução do fluxo de execução e a identificação de estratégias de evasão adotadas por cada binário.

¹<https://bazaar.abuse.ch>

7.0.3 Estrutura da Apresentação dos Resultados

As subseções seguintes apresentam os principais achados obtidos a partir dos módulos de rastreamento da *Contradef*. Cada técnica identificada é analisada individualmente, destacando como os dados coletados permitiram reconhecer indícios compatíveis com as técnicas descritas.

Cada resultado é discutido à luz dos eventos registrados e dos indícios extraídos durante a execução, ressaltando as estratégias adotadas pelas amostras para dificultar a análise e os meios pelos quais a *Contradef* conseguiu expor tais comportamentos.

7.1 Técnicas Evasivas Encontradas no *Obsidium*

De acordo com os dados reportados pelo *VirusTotal*, a amostra 0d0617fc515c...², é um executável malicioso de 64 bits para *Windows*. A análise de comportamento revela que sua principal função é atuar como um *cryptominer* (minerador de criptomoedas), especificamente o *XMRig*, que é usado para minerar *Monero*. Para evitar a detecção e garantir sua operação, o *malware* emprega várias técnicas de evasão: ele adiciona exclusões ao *Microsoft Defender* usando comandos do *PowerShell*, tenta adotar atrasos/anti-análise, faz reconhecimento do ambiente consultando comandos de processos, cria uma tarefa agendada (*schtasks*) para persistência, e abusa de um componente legítimo da *Microsoft* (*vbc.exe*, o compilador de *Visual Basic*) para executar suas operações de mineração. A primeira aparição (primeira vez enviado para análise) foi em 13/04/2023, e a última aparição (última submissão) em 14/04/2023.

7.1.1 Técnica anti-*debug*: detecção de padrões nos títulos de janelas via *EnumWindows*

O *Obsidium* emprega uma rotina de varredura dos títulos de janelas para identificar a presença de depuradores. O fluxo consiste em: (i) enumerar janelas com *EnumWindows*; (ii) filtrar janelas visíveis com *IsWindowVisible*; (iii) recuperar o par *thread/process* com *GetWindowThreadProcessId*; e (iv) obter o título com *InternalGetWindowText/GetWindowTextW*, sobre o qual são feitas buscas por padrões das ferramentas de depuração *OllyDbg*, *x64_dbg*, *WinDbg* e *IDA*. A Listagem 7.1 ilustra esse padrão de chamadas capturado no *TraceFncCall*: as linhas 1–4 mostram uma iteração completa *Enumeração* → *Filtro* → *Mapeamento* → *Leitura de título*. A sequência de chamadas → *IsWindowVisible* → *GetWindowThreadProcessId* → *InternalGetWindowText* se repete durante a enumeração de janelas.

Listagem 7.1: Sequência de chamadas de API para enumeração e filtragem de janelas

```
1 7ff802b61de0 T[0] C:\Windows\System32\USER32.dll : EnumWindows
2 7ff802b5e590 T[0] C:\Windows\System32\USER32.dll : IsWindowVisible
3 7ff802b42cd0 T[0] C:\Windows\System32\USER32.dll : GetWindowThreadProcessId
4 7ff802b685d0 T[0] C:\Windows\System32\USER32.dll : InternalGetWindowText
```

Do ponto de vista de memória, a rotina opera por comparação caractere a caractere entre o título retornado e um bloco interno de *strings* com nomes de depuradores. A Listagem 7.2 (extraída do *TraceMemory*) evidencia esse comportamento: na linha 1, um caractere do padrão "OllyDbg" é carregado de `[r10+rcx+1]` para AL; logo, ele é comparado (linha 2) ao correspondente do título de janela. Como não há correspondência, o laço avança para o próximo padrão, "x64_dbg" (linhas 5–6), depois "WinDbg" (linhas 9–10) e, por fim, "IDA" (linhas 13–14). Esse encadeamento confirma uma varredura sequencial sobre uma lista interna de padrões de texto.

²<https://www.virustotal.com/gui/file/0d0617fc515cd65ce750969d86f35563645173137d55b740cfd62163bc00cd6b>

Listagem 7.2: Rastreamento de memória: comparação de nomes de depuradores com o título de cada janela

```

1 [T0] 0x0ced426 | mov al, byte ptr [r10+rcx*1+0x1] | [Read 1 Op]: [0x0ced514] = 0x4f (79) ->
   "OllyDbg"
2 [T0] 0x0ced43f | cmp ax, word ptr [rbp+rcx*2] | [Read 1 Op]: [0x10ff7d0] = 0x0041 (65)
   -> "Administrador: Prompt..."
3 [T0] 0x0ced467 | movzx eax, byte ptr [r10] | [Read 1 Op]: [0x0ced513] = 0x07 (7)
4 [T0] 0x0ced47d | cmp byte ptr [r10], 0x0 | [Read 1 Op]: [0x0ced51c] = 0x07 (7)
5 [T0] 0x0ced426 | mov al, byte ptr [r10+rcx*1+0x1] | [Read 1 Op]: [0x0ced51d] = 0x78 (120) ->
   "x64_dbg"
6 [T0] 0x0ced43f | cmp ax, word ptr [rbp+rcx*2] | [Read 1 Op]: [0x10ff7d0] = 0x0041 (65)
   -> "Administrador: Prompt..."
7 [T0] 0x0ced467 | movzx eax, byte ptr [r10] | [Read 1 Op]: [0x0ced51c] = 0x07 (7)
8 [T0] 0x0ced47d | cmp byte ptr [r10], 0x0 | [Read 1 Op]: [0x0ced525] = 0x06 (6)
9 [T0] 0x0ced426 | mov al, byte ptr [r10+rcx*1+0x1] | [Read 1 Op]: [0x0ced526] = 0x57 (87) ->
   "WinDbg"
10 [T0] 0x0ced43f | cmp ax, word ptr [rbp+rcx*2] | [Read 1 Op]: [0x10ff7d0] = 0x0041 (65)
   -> "Administrador: Prompt..."
11 [T0] 0x0ced467 | movzx eax, byte ptr [r10] | [Read 1 Op]: [0x0ced525] = 0x06 (6)
12 [T0] 0x0ced47d | cmp byte ptr [r10], 0x0 | [Read 1 Op]: [0x0ced52d] = 0x04 (4)
13 [T0] 0x0ced426 | mov al, byte ptr [r10+rcx*1+0x1] | [Read 1 Op]: [0x0ced52e] = 0x49 (73) ->
   "IDA "
14 [T0] 0x0ced43f | cmp ax, word ptr [rbp+rcx*2] | [Read 1 Op]: [0x10ff7d0] = 0x0041 (65)
   -> "Administrador: Prompt..."

```

O trecho de código capturado com o `TraceInstructions`, apresentado na Listagem 7.3, mostra que o laço que se repete por cada padrão de texto inicia com a leitura do próximo caractere do padrão (linha 1, `mov al, [r10+rcx*1]`), onde `0x4f` → "O" do texto "OllyDbg", seguida por um teste simples de terminador (linha 3, `test al, al` e linha 5, `jz`) e pela comparação efetiva com o título (linha 7, `cmp ax, [rbp+rcx*2]`). As ramificações (linhas 8–9) conduzem ao avanço do laço ou ao desvio para a rotina de 'correspondência/não correspondência', de acordo com o resultado.

Listagem 7.3: Sequência de instruções para comparação caractere a caractere

```

1 0x0ced426 | mov al, byte ptr [r10+rcx*1+0x1] | [T0] al = 0x000000000000004f
2 0x0ced42b | jmp 0xced431 | [T0]
3 0x0ced431 | test al, al | [T0] al = 0x4f -> "'O'", al = 0x4f -> "'O'"
4 0x0ced433 | jmp 0xced436 | [T0]
5 0x0ced436 | jz 0xced4a8 | [T0]
6 0x0ced438 | jmp 0xced43f | [T0]
7 0x0ced43f | cmp ax, word ptr [rbp+rcx*2] | [T0] ax = 0x004f -> "O", memval = 0x10ff7d0
   -> 0x0041 -> "Administrador: Prompt..."
8 0x0ced444 | jmp 0xced44b | [T0]
9 0x0ced44b | jnz 0xced462 | [T0]

```

A relação entre módulos da *Contradef* evidencia: (i) a Listagem 7.1 do `TraceFncCall` documenta o padrão de chamadas `EnumWindows` → `IsWindowVisible` → `GetWindowThreadProcessId` → `InternalGetWindowText`, que antecede a comparação; (ii) a Listagem 7.2 do `TraceMemory` confirma, por *trace* de memória, a leitura sequencial dos padrões de texto internos e a comparação com o título corrente (textos para comparação são lidos da memória nas linhas 1–2, 5–6, 9–10 e 13–14); e (iii) a Listagem 7.3 do `TraceInstructions` expõe o laço de comparação que implementa o teste caractere a caractere. Em conjunto, as evidências sustentam que, para cada janela visível, o título é obtido e confrontado contra uma lista de assinaturas; uma correspondência positiva aciona o ramo anti-*debug*, alterando o fluxo de execução, desviando para a rotina de falha exibindo o texto "Debugger detected - please disable it and restart the application" e encerrando a execução.

A técnica é simples e eficaz, ela explora a presença de ferramentas de depuração no ambiente, mesmo quando não está diretamente anexada ao processo-alvo. Em termos de *trade-offs*, o método é propenso a poucos *falsos negativos*, mas pode gerar *falsos positivos* quando os padrões aparecem em títulos legítimos. Nos logs capturados, observou-se a leitura e comparação do

título do próprio console que executava a *ContradeF*, replicando com precisão o comportamento descrito nas Listagens 7.2 e 7.3.

Portanto, a análise aprofundada dos *traces* de execução permite concluir que o *Obsidium* implementa um algoritmo robusto que, para cada janela visível mesmo não pertencente ao seu próprio processo, extrai o título e o compara contra uma lista interna de assinaturas de ferramentas de análise. Uma correspondência positiva indica a presença de um ambiente de depuração, o que leva o programa a alterar seu fluxo de execução para evadir a análise.

Em resumo, a combinação de *TraceMemory*, *TraceInstructions* e *TraceFcnCall* permitiu reconstruir o mecanismo de varredura de títulos de janela e confirmar a presença de uma lista interna de depuradores alvo.

7.1.2 Técnica Anti-análise - Detecção de Overhead com a Função *GetTickCount*

A Listagem 7.4 ilustra o log do *FunctionInterceptor* que capturou a função *GetTickCount*, que retorna o tempo de execução do sistema em milissegundos desde o último *boot*, é comumente utilizada para detectar ambientes de análise, como um *debugger* ou uma *sandbox*. O uso dessa função de forma sequencial, como observado no resultado da *ContradeF*, onde foi invocada 6 vezes, indica uma tentativa de medir o tempo de execução e identificar se há um *overhead* significativo causado por ferramentas de análise.

Listagem 7.4: Invocações da função *GetTickCount*

```

1 [+] GetTickCount ...
2   Module name: C:\Windows\System32\KERNEL32.DLL
3   Thread: 0
4   Call Id: 0
5   Routine address: 7ff802935980
6   Return Value: 49693250
7 [*] Completed
8 [+] GetTickCount ...
9   Module name: C:\Windows\System32\KERNEL32.DLL
10  Thread: 0
11  Call Id: 1
12  Routine address: 7ff802935980
13  Return Value: 49693265
14 [*] Completed
15 ...

```

Quando o *Obsidium* faz essas invocações consecutivas, ele está verificando o tempo que decorre entre as chamadas. Se o tempo entre as invocações for maior do que o esperado, isso pode ser um indicativo de que está sendo executado em um ambiente monitorado, onde o *overhead* da análise afeta o tempo de execução. Esse comportamento sugere o uso de técnicas evasivas para detectar a presença de ferramentas de análise, como a depuração ou DBI, que podem introduzir atrasos na execução do código.

No caso específico do resultado apresentado, o *Obsidium* está comparando o valor de retorno da função entre cada invocação para detectar se há um aumento anormal no tempo de execução, sugerindo a presença de uma ferramenta de análise. Se a diferença entre o valor inicial de *GetTickCount* e o valor final for maior do que o esperado, isso pode ser interpretado como uma indicação de que está sendo analisado, levando-o a modificar seu comportamento ou encerrar a execução.

Listagem 7.5: Detecção de *Overhead* usando a função *GetTickCount*

1 0x000000cea2b9	0d0617f.exe	jmp r10	[T0]
2 0x7ff802935980	GetTickCount	mov ecx, 0x7ffe0320	[T0] ecx = 0x7ffe0320 (2147353376)
3 ...	...	...	
4 0x7ff802935997	GetTickCount	ret	[T0] rsp = 0x10ffb40

5 0x000000cebd8e	0d0617f.exe	jmp 0xcebd94	[T0]	
6 0x000000cebd94	0d0617f.exe	mov ebx, eax	[T0]	ebx = 0x02f64242 (49693250)
7 0x000000cebd96	0d0617f.exe	jmp 0xcebd9a	[T0]	
8 0x000000cebd9a	0d0617f.exe	jmp 0xcebd9f	[T0]	
9 0x000000cebd9f	0d0617f.exe	call r13	[T0]	rsp = 0x10ffb38
10 0x7ff802935980	GetTickCount	mov ecx, 0x7ffe0320	[T0]	ecx = 0x7ffe0320 (2147353376)
11 ...	...	...		
12 0x7ff802935997	GetTickCount	ret	[T0]	rsp = 0x10ffb40
13 0x000000cebd9a	0d0617f.exe	jmp 0xcebda9	[T0]	
14 0x000000cebd9a	0d0617f.exe	call r13	[T0]	rsp = 0x10ffb38
15 0x7ff802935980	GetTickCount	mov ecx, 0x7ffe0320	[T0]	ecx = 0x7ffe0320 (2147353376)
16 ...	...	...		
17 0x7ff802935997	GetTickCount	ret	[T0]	rsp = 0x10ffb40
18 0x000000cebdac	0d0617f.exe	jmp 0xcebdb3	[T0]	
19 0x000000cebdb3	0d0617f.exe	call r13	[T0]	rsp = 0x10ffb38
20 0x7ff802935980	GetTickCount	mov ecx, 0x7ffe0320	[T0]	ecx = 0x7ffe0320 (2147353376)
21 ...	...	...		
22 0x7ff802935997	GetTickCount	ret	[T0]	rsp = 0x10ffb40
23 0x000000cebdb6	0d0617f.exe	jmp 0xcebdba	[T0]	
24 0x000000cebdba	0d0617f.exe	call r13	[T0]	rsp = 0x10ffb38
25 0x7ff802935980	GetTickCount	mov ecx, 0x7ffe0320	[T0]	ecx = 0x7ffe0320 (2147353376)
26 ...	...	...		
27 0x7ff802935997	GetTickCount	ret	[T0]	rsp = 0x10ffb40
28 0x000000cebdbd	0d0617f.exe	jmp 0xcebdc4	[T0]	
29 0x000000cebdbc	0d0617f.exe	call r13	[T0]	rsp = 0x10ffb38
30 0x7ff802935980	GetTickCount	mov ecx, 0x7ffe0320	[T0]	ecx = 0x7ffe0320 (2147353376)
31 ...	...	...		
32 0x7ff802935997	GetTickCount	ret	[T0]	rsp = 0x10ffb40
33 0x000000cebdcc	0d0617f.exe	jmp 0xcebdcc	[T0]	
34 0x000000cebdcc	0d0617f.exe	mov edx, eax	[T0]	edx = 0x02f64280 (49693312)
35 0x000000cebdce	0d0617f.exe	jmp 0xcebdd5	[T0]	
36 0x000000cebdd5	0d0617f.exe	xor ecx, ecx	[T0]	rcx = 0, rflags = 0x246
37 0x000000cebdd7	0d0617f.exe	jmp 0xcebddb	[T0]	
38 0x000000cebddb	0d0617f.exe	sub edx, ebx	[T0]	rdx = 0x3e, rflags = 0x212
39 0x000000cebddb	0d0617f.exe	jmp 0xcebde3	[T0]	
40 0x000000cebde3	0d0617f.exe	mov eax, 0x1d0e	[T0]	eax = 0x1d0e (7438)
41 0x000000cebde8	0d0617f.exe	jmp 0xcebde8	[T0]	
42 0x000000cebde8	0d0617f.exe	cmp edx, 0x5	[T0]	edx = 0x003e (62), immval = 0x05
(5), rflags = 0x212				
43 0x000000cebdff0	0d0617f.exe	jmp 0xcebdf6	[T0]	
44 0x000000cebdff6	0d0617f.exe	cmovnz eax, ecx	[T0]	eax = 0x1d0e (7438), ecx = 0x0000
(0), rax = 0x1d0e				
45 0x000000cebdff9	0d0617f.exe	jmp 0xcebdfc	[T0]	
46 0x000000cebdffc	0d0617f.exe	jmp 0xcebe02	[T0]	
47 0x000000cebe02	0d0617f.exe	add rsp, 0x50	[T0]	rsp = 0x10ffb90, rflags = 0x206
48 0x000000cebe06	0d0617f.exe	pop rbx	[T0]	rbx = 0, rsp = 0x10ffb98
49 0x000000cebe07	0d0617f.exe	pop r13	[T0]	r13 = 0, rsp = 0x10ffba0
50 0x000000cebe09	0d0617f.exe	pop r12	[T0]	r12 = 0xcd8d5b, rsp = 0x10ffba8
51 0x000000cebe0b	0d0617f.exe	ret	[T0]	rsp = 0x10ffbb0

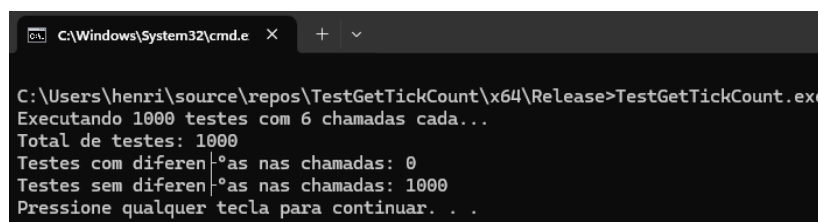
A Listagem 7.5 apresenta um trecho do código em *assembly* capturado pelo *TraceInstructions* que foi utilizado pelo *Obsidium* para detectar *overhead* através da função *GetTickCount*. Esse método realiza diversas chamadas consecutivas à função para medir o tempo do sistema, armazenando os valores retornados em registradores. Na instrução `sub edx, ebx` (linha 38), `ebx` contém o valor anterior de `eax`, que corresponde ao tempo retornado pela primeira chamada a *GetTickCount* (linha 6). Como resultado dessa instrução (linha 6), o registrador `edx` armazena a diferença entre os tempos calculados (`edx = edx - ebx`). Essas diferenças são posteriormente comparadas com um limiar fixo, neste caso, o valor 5, representado na instrução `cmp edx, 0x5` (linha 42). Caso a diferença calculada seja diferente de 0x5, o código pode alterar seu fluxo de execução, interromper a execução, adicionar verificações adicionais ou desviar o comportamento esperado, dependendo das condições estabelecidas no restante do programa.

A lógica por trás da comparação com um valor absoluto, como 0x5, não é imediatamente evidente, uma vez que, em verificações de *overhead*, seria mais comum avaliar se a diferença excede um limite predefinido. Levantamos duas hipóteses para justificar essa abordagem. A primeira hipótese sugere que a intenção seja confundir analistas. Nesse caso, o valor exato (0x5) não teria relevância para a funcionalidade do programa, mas sim o propósito de dificultar a

análise.

A segunda hipótese considera a detecção de ferramentas de análise específicas, como depuradores ou instrumentadores, que tendem a introduzir uma latência constante entre chamadas consecutivas a *GetTickCount*. Ferramentas dessa natureza podem acrescentar incrementos consistentes, como 1 ms, entre cada chamada, resultando em um valor acumulado constante, como 0x5, após cinco chamadas consecutivas.

Para avaliar o comportamento temporal entre chamadas consecutivas à função *GetTickCount*, implementamos um programa de teste que replicou o mesmo procedimento observado no software analisado. Em um ambiente de execução normal, executando em *bare metal*, o tempo entre chamadas não apresentou nenhuma diferença, e foi de 0 ms em 1000 execuções, cada uma com 6 chamadas sequenciais, como mostrado na Figura 7.1.



```
C:\Windows\System32\cmd.exe
C:\Users\henri\source\repos\TestGetTickCount\x64\Release>TestGetTickCount.exe
Executando 1000 testes com 6 chamadas cada...
Total de testes: 1000
Testes com diferenças nas chamadas: 0
Testes sem diferenças nas chamadas: 1000
Pressione qualquer tecla para continuar. . .
```

Figura 7.1: Teste de chamadas recorrentes à função *GetTickCount*

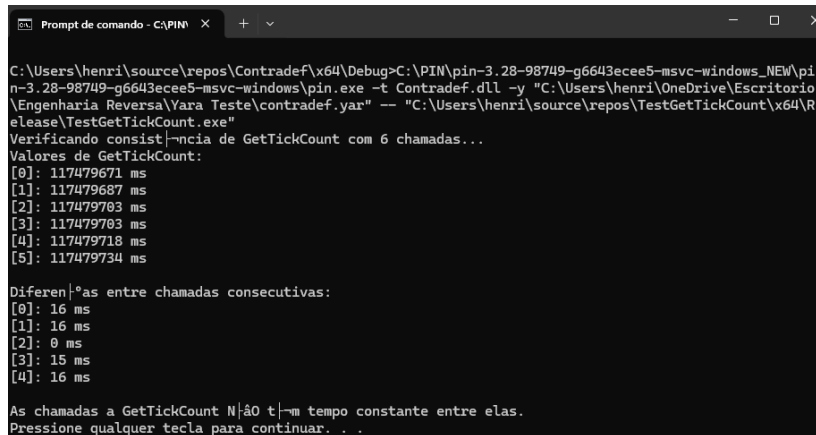
Testamos o programa de teste nas seguintes ferramentas: *WinDbg*, *x64dbg*, *IDA Freeware*, *OllyDbg*, *VirtualBox* e *Sandboxie*. Em todos os casos, a diferença de tempo entre as chamadas à função foi de 0 ms, sugerindo que as mesmas, em condições normais, com a execução contínua e sem paradas, não acrescentam *overhead* entre as chamadas à *GetTickCount*. Dessa forma, não foi possível identificar nenhuma dessas ferramentas como responsável por introduzir o atraso constante de 1 ms em cada chamada à função *GetTickCount*.

Por outro lado, ao executar o mesmo programa utilizando o instrumentador PIN, verificamos a introdução de um *overhead* variável em cada execução, conforme ilustrado nas Figuras 7.2 e 7.3. Esses resultados sugerem que o *overhead* variável detectado no ambiente instrumentado é uma característica específica da ferramenta PIN, destacando seu impacto temporal em execuções monitoradas.

Esses resultados confirmam que, embora o comportamento observado no *Obsidium* seja capaz de identificar a presença de ferramentas de análise ou instrumentação por meio da técnica de medição temporal, não foi possível determinar para qual ferramenta específica essa técnica foi projetada.

Por outro lado, o código faz uso de chamadas indiretas via registradores `jmp r13` e `call r13` (linhas 1, 9, 14, 19, 24 e 29), para dificultar o rastreamento de fluxo. O uso de saltos incondicionais *jmp* intercalados demonstra o uso de métodos para dificultar a análise, como ofuscação.

Essas práticas são indicativas de uma técnica evasiva projetada para detectar alguma ferramenta específica que usa ambientes controlados, como sandboxes ou depuradores. Essa técnica, além de tentar obscurecer o objetivo final do código frustrando análises automáticas ou estáticas, também tenta identificar uma discrepância específica no comportamento temporal, característico de ambientes monitorados. Assim, o código exemplifica uma abordagem sofisticada de evasão, misturando ofuscação e análise ambiental para ocultar sua verdadeira intenção.



```

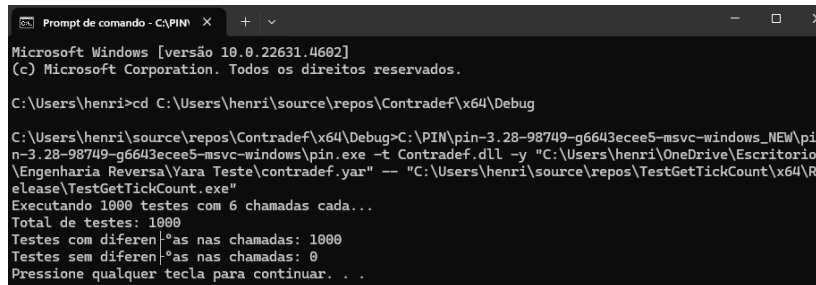
C:\Users\henri\source\repos\Contradef\x64\Debug>C:\PIN\pin-3.28-98749-g6643ecee5-msvc-windows_NEW\pin-3.28-98749-g6643ecee5-msvc-windows\pin.exe -t Contradef.dll -y "C:\Users\henri\OneDrive\Escritorio\Engenharia Reversa\Vara Teste\contradef.yar" -- "C:\Users\henri\source\repos\TestGetTickCount\x64\Release\TestGetTickCount.exe"
Verificando consistência de GetTickCount com 6 chamadas...
Valores de GetTickCount:
[0]: 117479671 ms
[1]: 117479687 ms
[2]: 117479703 ms
[3]: 117479703 ms
[4]: 117479718 ms
[5]: 117479734 ms

Diferenças entre chamadas consecutivas:
[0]: 16 ms
[1]: 16 ms
[2]: 0 ms
[3]: 15 ms
[4]: 16 ms

As chamadas a GetTickCount não são feitas em tempo constante entre elas.
Pressione qualquer tecla para continuar. . .

```

Figura 7.2: Teste de execução da função *GetTickCount* invocada 6 vezes sendo instrumentada pelo o Pin



```

Microsoft Windows [versão 10.0.22631.4602]
(c) Microsoft Corporation. Todos os direitos reservados.

C:\Users\henri>cd C:\Users\henri\source\repos\Contradef\x64\Debug

C:\Users\henri\source\repos\Contradef\x64\Debug>C:\PIN\pin-3.28-98749-g6643ecee5-msvc-windows_NEW\pin-3.28-98749-g6643ecee5-msvc-windows\pin.exe -t Contradef.dll -y "C:\Users\henri\OneDrive\Escritorio\Engenharia Reversa\Vara Teste\contradef.yar" -- "C:\Users\henri\source\repos\TestGetTickCount\x64\Release\TestGetTickCount.exe"
Executando 1000 testes com 6 chamadas cada...
Total de testes: 1000
Testes com diferenças nas chamadas: 1000
Testes sem diferenças nas chamadas: 0
Pressione qualquer tecla para continuar. . .

```

Figura 7.3: Teste de execução de 1000 iterações da função *GetTickCount* invocada 6 vezes sendo instrumentada pelo o Pin

7.1.3 Técnica Anti-debug - Detecção de Depuradores com a Função *RaiseException*

Além das estratégias de detecção de depuradores por meio de verificação de janelas ou nomes de processos, o *Obsidium* também emprega a técnica de detecção baseada em exceções. De acordo com Sikorski and Honig (2012), a manipulação de exceções é capaz de interromper ou identificar a presença de um *debugger*, uma vez que ferramentas de depuração geralmente interceptam as exceções antes de repassá-las ao processo. Se o *debugger* não encaminha corretamente a exceção, a falha no fluxo de tratamento pode ser detectada pelo próprio processo instrumentado, denunciando o ambiente de análise.

A ferramenta *Contradef* registrou por meio do módulo *FunctionInterceptor* duas chamadas à função *RaiseException*, conforme apresentado na Listagem 7.6.

Listagem 7.6: Invocações da função *RaiseException* para detecção de depuradores

```

1 ...
2 [+] RaiseException ...
3   Thread: 0
4   Call Id: 0
5   Routine address: 0x7ff800af5300
6   Parameters:
7     dwExceptionCode: 0x40010006

```

```

8         dwExceptionFlags: 0x0
9         nNumberOfArguments: 2
10        lpArguments: 0x10ffb60
11        Address of calling function: 0xceae8c
12        Argument[0]: 0x0
13        Argument[1]: 0xceaf6e
14        [-] Start of call RaiseException
15        Returned from RaiseException: (VOID)
16        [-] Call from RaiseException completed
17        [*] Completed
18        [+] RaiseException ...
19        Thread: 0
20        Call Id: 1
21        Routine address: 0x7ff800af5300
22        Parameters:
23        dwExceptionCode: 0x4001000a
24        dwExceptionFlags: 0x0
25        nNumberOfArguments: 4
26        lpArguments: 0x10ffb60
27        Address of calling function: 0xceaece
28        Argument[0]: 0x0
29        Argument[1]: 0xceaf6e
30        Argument[2]: 0x0
31        Argument[3]: 0xceaf6e
32        [-] Start of call RaiseException
33        Returned from RaiseException: (VOID)
34        [-] Call from RaiseException completed
35        [*] Completed
36        ...

```

A função *RaiseException* dispara uma exceção no processo atual. Quando um *debugger* está anexado, o fluxo normal de tratamento de exceções é interrompido, possibilitando ao *software* identificar uma tentativa de depuração. Na Listagem 7.6, vemos que o *Obsidium* chama *RaiseException* com diferentes códigos de exceção (0x40010006 e 0x4001000a), o que sugere a verificação de diferentes cenários de manipulação de exceções.

Os valores 0x40010006 e 0x4001000a correspondem às constantes `DBG_PRINTEXCEPTION_C` e `DBG_PRINTEXCEPTION_WIDE_C`, respectivamente, definidas no arquivo de código *winnt.h* do *Windows SDK*. E, de acordo com [Check Point Research \(2022b\)](#), as funções de depuração, como *ntdll:DbgPrint* e *kernel32:OutputDebugStringW*, geram a exceção `DBG_PRINTEXCEPTION_C` (0x40010006). A Listagem 7.7 apresenta a implementação da técnica baseada no exemplo ilustrado por [Check Point Research \(2022b\)](#), que demonstra o funcionamento da técnica evasiva.

Listagem 7.7: Implementação da técnica de detecção de depuradores baseada no exemplo de [Check Point Research \(2022b\)](#)

```

1 bool IsDebugged() {
2     __try
3     {
4         RaiseException(DBG_PRINTEXCEPTION_C, 0, 0, 0);
5     }
6     __except (GetExceptionCode() == DBG_PRINTEXCEPTION_C)
7     {
8         return false;
9     }
10    return true;
11 }
12 int main() {
13     std::string isDbgd = IsDebugged() ? "TRUE" : "FALSE";
14     std::cout << "IsDebugged: " << isDbgd << std::endl;
15     system("pause");
16     return 0;
17 }

```

Ao compilar e executar o código da Listagem 7.7, obtém-se a saída apresentada na Figura 7.4, que ilustra o comportamento do programa em dois cenários distintos: janela da esquerda executando com um depurador anexado e janela da direita sem a presença de um depurador.

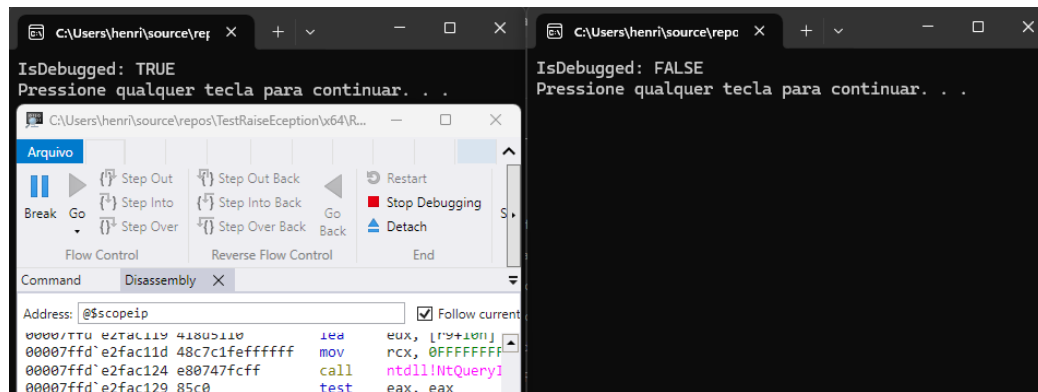


Figura 7.4: Execução da técnica de detecção de depuradores usando a função *RaiseException*

Na Listagem 7.6, é possível observar a geração deliberada de exceções via *RaiseException*. Esse padrão sugere que o *Obsidium* está testando ativamente como o ambiente lida com as exceções, seguindo a descrição de Sikorski and Honig (2012) de que a detecção baseada em exceções depende do fato de que os depuradores capturarão a exceção e não a passarão imediatamente para o processo que está sendo depurado para tratamento. A configuração padrão na maioria dos depuradores é capturar exceções e não passá-las para o programa. Se o depurador não passar a exceção para o processo corretamente, essa falha poderá ser detectada no mecanismo de tratamento de exceções do processo (Sikorski and Honig, 2012). Se um programa está sendo executado com um depurador anexado, então esse depurador lidará com a exceção. Entretanto, se não houver um depurador presente e um manipulador de exceção estiver registrado, essa exceção será capturada pelo manipulador de exceção. Se o depurador não encaminhar corretamente a exceção ao processo, o software sob análise poderá tomar conhecimento de sua presença.

No *trace* de instruções, comprovamos a rotina de verificação do *Obsidium* após a chamada à função *RaiseException*, apresentada na Listagem 7.8.

Listagem 7.8: Detecção de depuradores com a função *RaiseException*

1 0x0ceaf71	0d0617f.exe	sub rsp, 0x58	rsp = 0x10fef30
2 0x0ceaf75	0d0617f.exe	mov rdx, qword ptr [rcx]	rdx = 0x10ffa50 → 0x40010006
3 0x0ceaf78	0d0617f.exe	jmp 0xceaf7c	
4 0x0ceaf7c	0d0617f.exe	mov eax, dword ptr [rdx]	eax = 0x40010006
5 0x0ceaf7e	0d0617f.exe	jmp 0xceaf84	
6 0x0ceaf84	0d0617f.exe	cmp eax, 0x4001000a	eax = 0x40010006, immval = 0x4001000a
7 0x0ceaf89	0d0617f.exe	jmp 0xceaf8f	
8 0x0ceaf8f	0d0617f.exe	jz 0xceafa7	
9 0x0ceaf91	0d0617f.exe	jmp 0xceaf98	
10 0x0ceaf98	0d0617f.exe	cmp eax, 0x40010006	eax = 0x40010006, immval = 0x40010006
11 0x0ceaf9d	0d0617f.exe	jmp 0xceafa2	
12 0x0ceafa2	0d0617f.exe	jnz 0xceb01b	
13 0x0ceafa4	0d0617f.exe	jmp 0xceafa7	

O trecho de código da Listagem 7.8 apresenta o código após a chamada à *RaiseException*, com a verificação de códigos de exceção (0x4001000a e 0x40010006) que são usados pelo *Obsidium* para detectar e reagir à presença de um *debugger*. Na linha 6, o código compara *eax* com 0x4001000a. Se forem iguais (jz, linha 8), segue para o bloco de instruções que manipula essa exceção. De forma semelhante, na linha 10, ocorre a comparação de *eax* com 0x40010006. Se o valor não for igual (jnz, linha 12), a rotina de encerramento é acionada com a mensagem de depurador detectado. O uso de instruções *jmp* e *cmp* de maneira não linear e diferentes tipos de saltos (jz e jnz) contribui para dificultar a análise estática, tornando a detecção e a engenharia reversa mais complexas.

7.1.4 Técnica Anti-debug - Detecção do *TitanHide* (Ferramenta de Ocultação de Depuradores) com a Função *CreateFileW*

O log registrado pelo *FunctionInterceptor* da *Contradef* durante a execução do *Obsidium* também capturou uma chamada à função *CreateFileW* como apresentado na Listagem 7.9. Essa função é utilizada no *Windows* para abrir ou criar arquivos ou dispositivos ([Microsoft Learn, 2023a](#)).

Listagem 7.9: Invocação da função *CreateFileW* para detecção do *TitanHide*

```

1 ...
2 [+] CreateFileW ...
3   Thread: 0
4   Call Id: 0
5   Routine address: 7ff800afe44f
6   Parameters:
7       lpFileName: \\.\TitanHide
8       dwDesiredAccess: 0x80000000
9       dwShareMode: 1
10      dwCreationDisposition: 0
11      dwFlagsAndAttributes: 0
12   Address of calling function: 10ffb78
13 [-] Start of file access
14   Returned Handle: 0xffffffffffffffff
15 [-] File access completed
16 [*] Completed
17 ...

```

Nos parâmetros capturados, o parâmetro *lpFileName* contém o valor *\\.\TitanHide*, que faz referência a um *namespace* de dispositivo do *Win32*, em vez do *namespace* de arquivo ([Microsoft Learn, 2024d](#)). Geralmente, o prefixo *\\.* é utilizado para indicar que o sistema deve tratar o caminho como um dispositivo e não como um arquivo comum em disco. Nesse caso, o *Obsidium* tenta abrir ou se comunicar com um dispositivo chamado *TitanHide*, relacionado a um *driver* cujo propósito é ocultar depuradores de certos processos. Esse *driver* intercepta várias funções do kernel *Nt* e modifica os valores de retorno das funções originais ([Duncan Ogilvie, 2022](#)). Com o *TitanHide*, para ocultar um processo, é necessário passar uma estrutura simples com um *ProcessID* e as opções de ocultação a serem habilitadas para o *driver*.

O *TitanHide* ao ser instalado e configurado, é capaz de interceptar várias funções do *kernel Nt* e alterar os valores de retorno das funções originais ([Duncan Ogilvie, 2022](#)). Para ocultar um processo, basta passar uma estrutura simples contendo o *ProcessID* e as opções de ocultação desejadas ao *driver*.

O parâmetro de Acesso Desejado (*dwDesiredAccess*) contém um valor numérico que corresponde a constantes que definem o tipo de acesso solicitado. O valor *0x80000000* (hexadecimal) corresponde à constante *GENERIC_READ*, definida em *winnt.h*, indicando acesso genérico de leitura.

Segundo [Microsoft Learn \(2023a\)](#), o parâmetro *dwShareMode* com valor 1 corresponde à constante *FILE_SHARE_READ*, permitindo que outros processos leiam do mesmo arquivo ou dispositivo enquanto este permanece aberto.

Já o parâmetro *dwCreationDisposition* com valor 0 não corresponde a nenhum valor válido da enumeração esperada pela *CreateFileW*. Os valores válidos são: *CREATE_NEW* (1), *CREATE_ALWAYS* (2), *OPEN_EXISTING* (3), *OPEN_ALWAYS* (4) e *TRUNCATE_EXISTING* (5). Para dispositivos (caminhos no *namespace* *\\.* como *\\.\TitanHide*), o *wrapper* interno da função ajustou automaticamente esse parâmetro para *OPEN_EXISTING* (3) durante a chamada, pois é o padrão para dispositivos que não são arquivos ([Microsoft Learn, 2023a](#)).

O valor retornado pela função foi o *handle* *ffffffffffffffff* (−1 em notação decimal para ponteiros ou *handles*), o que sugere falha na chamada. Caso o *driver TitanHide* estivesse

instalado e configurado no *Windows*, um *handle* válido seria retornado (usual e diferenciado de -1). O retorno inválido (-1) implica que o *driver* **TitanHide** não está presente no sistema.

Listagem 7.10: Assembly que verifica o valor retornado pela função **CreateFileW** usada para detectar o **TitanHide**

1 0x0cec6e9	0d0617f.exe	jmp 0xcec6f0	[T0]	
2 0x0cec6f0	0d0617f.exe	cmp rax, 0xffffffffffffffff	[T0]	rax = 0xffffffffffffffff
3 0x0cec6f4	0d0617f.exe	jmp 0xcec6f8	[T0]	
4 0x0cec6f8	0d0617f.exe	jz 0xcec73e	[T0]	
5 0x0cec73e	0d0617f.exe	jmp 0xcec741	[T0]	
6 0x0cec741	0d0617f.exe	xor eax, eax	[T0]	rax = 0
7 0x0cec743	0d0617f.exe	jb 0xcec737	[T0]	
8 0x0cec745	0d0617f.exe	jmp 0xcec74b	[T0]	
9 0x0cec74b	0d0617f.exe	add rsp, 0x50	[T0]	rsp = 0x10ffbba0
10 0x0cec74f	0d0617f.exe	pop r12	[T0]	r12 = 0xcd8d5b
11 0x0cec751	0d0617f.exe	jmp 0xcec754	[T0]	
12 0x0cec754	0d0617f.exe	ret	[T0]	rsp = 0x10ffbb0

A Listagem 7.10 apresenta o trecho de código em *assembly* capturado pelo **TraceInstructions** após a chamada à função **CreateFileW**. Na instrução **cmp rax, 0xffffffffffffffff** (linha 2), o programa verifica se o valor retornado em **RAX** é equivalente a -1 , o que indicaria uma falha na obtenção do *handle* ao tentar acessar o dispositivo **\\.\TitanHide**.

A instrução **jz 0xcec73e** (linha 4) redireciona o fluxo caso **RAX** seja igual a $0xffff\dots$. Nesse cenário, entende-se que a chamada para abrir o dispositivo **\\.\TitanHide** falhou ou não obteve um *handle* válido.

Ao chegar em **0x0cec741** (linha 6), o programa executa **xor eax, eax**, que zera o registrador **EAX** (por extensão, **RAX** assume valor zero). Essa atribuição pode indicar que, caso o *handle* esteja inválido, o código define internamente um estado de “sucesso falso” ou “retorno padrão”, ajustando **RAX** para zero.

As instruções **add rsp, 0x50** (linha 9) e **pop r12** (linha 10) restauram parte do contexto da pilha, enquanto **ret** (linha 12) conclui a função. Esse conjunto sinaliza que o programa encerra a rotina após constatar a falha no *handle*, ou seja, a inexistência do dispositivo **TitanHide**.

Essa verificação corresponde a uma técnica de detecção de *drivers*, permitindo que o *Obsidium* identifique o uso de ferramentas como **TitanHide**, comumente empregadas para ocultar depuradores. Ao confirmar (ou não) a presença desse dispositivo, o *Obsidium* ajusta seu comportamento para encerrar a execução.

7.1.5 Técnica Anti-Debug – Detecção de Depuradores com a Função **NtQueryInformationProcess** e **ProcessDebugPort**

Além das estratégias de detecção de ferramentas como o **TitanHide**, o *Obsidium* também explora o parâmetro **ProcessDebugPort** (código 7 na *enum* **PROCESSINFOCLASS**) para identificar a presença de depuradores no processo. A Listagem 7.11 apresenta o registro da chamada à função **NtQueryInformationProcess** capturado pelo **FunctionInterceptor**, exibindo os parâmetros utilizados:

Listagem 7.11: Invocação da função **NtQueryInformationProcess** com **ProcessDebugPort** para detecção de depuradores

```

1 [+] NtQueryInformationProcess ...
2   Thread: 0
3   Call Id: 0
4   Routine address: 0x7ff8032ad800
5   Parameters:
6       ProcessHandle: 0xffffffffffffffff
7       ProcessInformationClass: 7
8       ProcessInformation: 0x10ffb60

```

```

9      ProcessInformationLength: 8
10     ReturnLength: 0x10ffb70
11     Address of calling function: 0xcec8d7
12 [-] Start of call NtQueryInformationProcess
13     Returned from NtQueryInformationProcess (NTSTATUS): 0x0
14     Successful operation (STATUS_SUCCESS).
15     (Read 256 bytes from ProcessInformation, displaying in hex): d7 c8 ce ...
16 [-] Call from NtQueryInformationProcess completed
17 [*] Completed

```

Conforme descrito em [Microsoft Learn \(2023b\)](#) e [Check Point Research \(2022a\)](#), os valores observados na Listagem 7.11 podem ser interpretados da seguinte forma:

- **ProcessHandle** = 0xfff... (*pseudohandle*): indica, geralmente, uma referência ao processo atual;
- **ProcessInformationClass** = 7: representa a consulta ao *ProcessDebugPort*, usada para obter informações relacionadas à porta de depuração do processo;
- **ProcessInformation** = 0x10ffb60: endereço de memória onde será armazenado o valor da porta de depuração (*debug port*);
- **ReturnLength** = 0x10ffb70: ponteiro para um endereço onde será escrita a quantidade de bytes retornados pela operação.

Como descrito por [Check Point Research \(2022a\)](#), caso um debugger esteja anexado ao processo, o *debug port* pode ser diferente de zero ou apresentar valor que indique a existência de um canal de depuração. Se não houver depurador presente, a função tende a retornar zero (ou outro valor) que sinalize a ausência de debug port. A Listagem 7.12, mostra o log capturado pelo *TraceInstructions*, que ilustra as instruções em *assembly* responsáveis por verificar o *NTSTATUS* retornado e o valor escrito na estrutura (*ProcessInformation*).

Listagem 7.12: Assembly que verifica o valor retornado pela função *NtQueryInformationProcess* com o parâmetro *ProcessDebugPort* usado para detectar o depuradores

1 0x0cea294	0d0617f.exe	mov rcx, qword ptr [rsp+0x70]	rcx = 0xffffffffffffffff
2 0x0cea299	0d0617f.exe	mov rdx, qword ptr [rsp+0x78]	rdx = 0x7
3 0x0cea29e	0d0617f.exe	mov r8, qword ptr [rsp+0x80]	r8 = 0x10ffb60 -> 0x0
4 0x0cea2a6	0d0617f.exe	mov r9, qword ptr [rsp+0x88]	r9 = 0x8
5 0x0cea2ae	0d0617f.exe	mov r10, r13	r10 = 0x00007ff8032ad800
6 0x0cea2b1	0d0617f.exe	add rsp, 0x58	rsp = 0x10ffaf8
7 0x0cea2b5	0d0617f.exe	pop r13	r13 = 0x7ff803210000
8 0x0cea2b7	0d0617f.exe	pop r12	r12 = 0xcd8d5b
9 0x0cea2b9	0d0617f.exe	jmp r10	
10 0x7ff8032ad800	ZwQuery...	mov r10, rcx	r10 = 0xffffffffffffffff
11 ...	...	...	
12 0x7ff8032ad814	ZwQuery...	ret	rsp = 0x10ffb10
13 0x0cec8d7	0d0617f.exe	jmp 0xcec8de	
14 0x0cec8de	0d0617f.exe	test eax, eax	eax = 0x0, eax = 0x0
15 0x0cec8e0	0d0617f.exe	jmp 0xcec8e7	
16 0x0cec8e7	0d0617f.exe	jnz 0xcec919	
17 0x0cec8e9	0d0617f.exe	jmp 0xcec8ee	
18 0x0cec8ee	0d0617f.exe	cmp dword ptr [rbp+0x10], 0x8	memval [0x010ffb70] -> 0x8,
		immval = 0x08	
19 0x0cec8f2	0d0617f.exe	jmp 0xcec8f9	
20 0x0cec8f9	0d0617f.exe	jnz 0xcec919	
21 0x0cec8fb	0d0617f.exe	jmp 0xcec902	
22 0x0cec902	0d0617f.exe	cmp qword ptr [rbp], 0x0	memval [0x10ffb60] -> 0x0,
		immval = 0x00	
23 0x0cec907	0d0617f.exe	jmp 0xcec90d	
24 0x0cec90d	0d0617f.exe	jnz 0xcecba4	
25 0x0cec913	0d0617f.exe	jmp 0xcec919	

As instruções `mov rcx, qword ptr [rsp+0x70]` (linha 1), `mov rdx, qword ptr [rsp+0x78]` (linha 2), entre outras, carregam os argumentos correspondentes a `ProcessHandle`, `ProcessInformationClass` e afins, antes de efetivamente chamar `NtQueryInformationProcess`.

Em `0x0cea2b9` (linha 9), o fluxo de execução salta (`jmp r10`) para o endereço `0x7ff8032ad800`, onde reside a função `ZwQueryInformationProcess` que é a implementação de `NtQueryInformationProcess` em `ntdll.dll`.

Após a chamada, em `0x0cec8de` (linha 14), ocorre a instrução `test eax, eax`, que verifica se a função retornou 0 (`STATUS_SUCCESS`) ou não. Em seguida, instruções como `jmp`, logo `jnz` que controla o desvio para diferentes rotas de execução, dependendo do valor.

A instrução `cmp dword ptr [rbp+0x10], 0x8` (linha 18) checa se o tamanho retornado (`ReturnLength`) corresponde ao valor esperado (8 bytes). Logo após, ocorre `cmp qword ptr [rbp], 0x0` (linha 22), que determina se o *debug port* efetivamente está presente ou não (0x0 indica ausência de debugger, enquanto valores diferentes podem sinalizar um depurador anexo).

Com o parâmetro `ProcessDebugPort`, o *Obsidium* identifica se existe um *debug port* associado ao processo. Em um ambiente de análise, com um depurador conectado, o *debug port* tende a ser diferente de zero. Se a verificação confirma a existência de um debug port, o *Obsidium* pode reagir encerrando a execução ou alterando o fluxo do programa para evitar a análise. Como destacado em outras técnicas *anti-debug*, essa estratégia é usada de forma redundante em conjunto com outras, dificultando o trabalho de analistas e de ferramentas de análise.

7.1.6 Técnica Anti-Debug – Detecção de Depuradores com a Função `NtQueryInformationProcess` e `ProcessDebugObjectHandle`

O *Obsidium* também se vale da chamada à função `NtQueryInformationProcess` para identificar a presença de depuradores. Segundo [Check Point Research \(2022a\)](#), essa técnica faz uso do parâmetro `ProcessDebugObjectHandle`, definido pelo valor 30 (0x1e) na *enum* `PROCESSINFOCLASS` não documentada oficialmente, conforme ilustrado na Listagem 7.13.

Listagem 7.13: Invocação da função `NtQueryInformationProcess` com `ProcessDebugObjectHandle` para detecção de depuradores

```

1 [+] NtQueryInformationProcess ...
2   Thread: 0
3   Call Id: 2
4   Routine address: 0x7ff8032ad800
5   Parameters:
6     ProcessHandle: 0xffffffffffffffff
7     ProcessInformationClass: 30
8     ProcessInformation: 0x10ffb68
9     ProcessInformationLength: 8
10    ReturnLength: 0x10ffb70
11   Address of calling function: 0xcec972
12 [-] Start of call NtQueryInformationProcess
13   Returned from NtQueryInformationProcess (NTSTATUS): 0xc0000353
14   Operation failed (NTSTATUS != 0). Code: 0xc0000353
15 [-] Call from NtQueryInformationProcess completed
16 [*] Completed

```

Como é possível observar no trecho extraído do `FunctionInterceptor` (Listagem 7.13), o *Obsidium* chama `NtQueryInformationProcess` passando:

- `ProcessHandle` = `0xffffffffffffffff`, o que normalmente se refere ao processo atual (pseudohandle),
- `ProcessInformationClass` = 30, associado ao valor `ProcessDebugObjectHandle`,

- `ProcessInformation = 0x10ffb68`: endereço de memória onde será armazenado o valor do *debug object*;
- `ProcessInformationLength = 8`, para armazenar um `HANDLE` ou ponteiro para o *debug object* retornado,
- `ReturnLength`, um ponteiro para onde será escrita a quantidade de bytes realmente retornados.

Nesse contexto, o *Obsidium* busca obter o *handle* (`HANDLE`) do objeto de depuração associado ao processo. Se esse *debug object* estiver presente, significa que um depurador pode estar anexado ao processo. Caso contrário, a função deve retornar um status de erro (por exemplo, `0xc0000353`, que indica `STATUS_OBJECT_NAME_NOT_FOUND` ou outro código de falha dependendo da implementação).

A Listagem 7.14 exibe as instruções em *assembly* que verificam o retorno da função *NtQueryInformationProcess*. O valor armazenado em `EAX` ou `RAX` (dependendo do modo de execução) é comparado com zero ou outra condição, determinando o fluxo de execução.

Listagem 7.14: Assembly que verifica o valor retornado pela função *NtQueryInformationProcess* com o parâmetro `ProcessDebugObjectHandle` usado para detectar o depuradores

1	0x0cec972		0d0617f.exe		jmp 0xcec976		
2	0x0cec976		0d0617f.exe		test eax, eax		eax = 0xc0000353, eax = 0xc0000353
3	0x0cec978		0d0617f.exe		jo 0xcec98a		
4	0x0cec97a		0d0617f.exe		jnz 0xcec9a0		
5	0x0cec9a0		0d0617f.exe		jmp 0xcec9a4		

A instrução `test eax, eax` (linha 2) avalia se o valor retornado (no registrador `EAX`) é zero ou diferente de zero. Em caso de erro (por exemplo, `0xc0000353`), o fluxo desvia para `0xcec9a0`, o que sinaliza que não foi obtido um *debug object* válido.

Caso o retorno fosse bem-sucedido (por exemplo, `STATUS_SUCCESS = 0`), significaria que foi retornado um *debug object* associado ao processo, indicando a provável presença de um depurador. Nessa situação, o *Obsidium* poderia reagir interrompendo a execução ou alterando seu comportamento para evitar a análise.

Conforme a instrução `jnz` (linha 4), se o valor de `EAX` for diferente de zero, o código pula para um bloco de tratamento de erro ou comportamento evasivo. Esse tipo de verificação permite ao *Obsidium* decidir como proceder na eventualidade de detectar um *debugger*.

Essa técnica *anti-debug* busca verificar se há um objeto de depuração atribuído ao processo atual, o que geralmente ocorre quando um *debugger* está anexado. Como observado por [Check Point Research \(2022b\)](#), a detecção de um *debugger* pode levar o *malware* ou o protetor (como o *Obsidium*) a alterar seu comportamento, dificultando a análise.

7.1.7 Técnica Anti-DBI/Debug – Detecção de Depuradores com a Função *NtQueryInformationProcess* e *ProcessDebugFlags*

Além de verificar parâmetros como `ProcessDebugObjectHandle` (`0x1E`) e `ProcessDebugPort` (`0x07`), o *Obsidium* também emprega a classe `0x1F` – *ProcessDebugFlags* – ao chamar *NtQueryInformationProcess* para identificar a presença de depuradores. A Listagem 7.15 ilustra o registro dessa chamada, enquanto a Listagem 7.16 detalha o fluxo de instruções em *assembly* que interpreta o valor retornado.

Listagem 7.15: Invocação da função *NtQueryInformationProcess* com `ProcessDebugFlags` para detecção de depuradores

```

1 [+] NtQueryInformationProcess ...
2   Thread: 0
3   Call Id: 4
4   Routine address: 0x7ff8032ad800
5   Parameters:
6     ProcessHandle: 0xffffffffffffffff
7     ProcessInformationClass: 31
8     ProcessInformation: 0x10ffb74
9     ProcessInformationLength: 4
10    ReturnLength: 0x10ffb70
11   Address of calling function: 0xceca06
12 [-] Start of call NtQueryInformationProcess
13   Returned from NtQueryInformationProcess (NTSTATUS): 0x0
14   Successful operation (STATUS_SUCCESS).
15   (Read 256 bytes from ProcessInformation, displaying in hex): 6 ca ce ...
16 [-] Call from NtQueryInformationProcess completed
17 [*] Completed

```

Os valores observados na Listagem 7.15 podem ser interpretados da seguinte forma:

- `ProcessHandle = 0xffff...`: *Pseudohandle* que normalmente se refere ao processo atual.
- `ProcessInformationClass = 31`: Corresponde ao parâmetro *ProcessDebugFlags*.
- `ProcessInformation = 0x10ffb74`: Endereço onde o valor de *ProcessDebugFlags* será escrito.
- `ProcessInformationLength = 4`: Define o tamanho esperado (em bytes) para armazenar o valor retornado.
- `ReturnLength = 0x10ffb70`: Ponteiro para a quantidade de bytes efetivamente retornados pela função.

De acordo com [Check Point Research \(2022a\)](#), a estrutura `EPROCESS` do núcleo (*kernel*) do Windows contém o campo *NoDebugInherit*, cujo valor inverso pode ser obtido por meio da classe *ProcessDebugFlags* (0x1F), não documentada oficialmente. Se o valor retornado for 0, significa que há um depurador presente. Em outras situações, se não há depurador anexado, o valor costuma ser diferente de zero.

A Listagem 7.16 apresenta o fluxo de instruções responsável por chamar *NtQueryInformationProcess* e analisar o resultado:

Listagem 7.16: Assembly que verifica o valor retornado pela função *NtQueryInformationProcess* com o parâmetro *ProcessDebugFlags* usado para detectar o depuradores

1 0x0cea294	0d0617f.exe	mov rcx, qword ptr [rsp+0x70]	rcx = 0xffffffffffffffff
2 0x0cea299	0d0617f.exe	mov rdx, qword ptr [rsp+0x78]	rdx = 0x1f
3 0x0cea29e	0d0617f.exe	mov r8, qword ptr [rsp+0x80]	r8 = 0x10ffb74 → 0x1 (1)
4 0x0cea2a6	0d0617f.exe	mov r9, qword ptr [rsp+0x88]	r9 = 0x4
5 0x0cea2ae	0d0617f.exe	mov r10, r13	r10 = 0x00007ff8032ad800
6 0x0cea2b1	0d0617f.exe	add rsp, 0x58	rsp = 0x10ffa8
7 0x0cea2b5	0d0617f.exe	pop r13	r13 = 0x7ff803210000
8 0x0cea2b7	0d0617f.exe	pop r12	r12 = 0xcd8d5b
9 0x0cea2b9	0d0617f.exe	jmp r10	
10 0x7ff8032ad800	ZwQuery...	mov r10, rcx	r10 = 0xffffffffffffffff
11 ...	...	...	
12 0x7ff8032ad814	ZwQuery...	ret	rsp = 0x10ffb10
13 0x0ceca06	0d0617f.exe	jmp 0xceca0b	
14 0x0ceca0b	0d0617f.exe	test eax, eax	eax = 0x0, eax = 0x0
15 0x0ceca0d	0d0617f.exe	jnb 0xceca14	
16 0x0ceca14	0d0617f.exe	jnz 0xceca41	
17 0x0ceca16	0d0617f.exe	jmp 0xceca1d	
18 0x0ceca1d	0d0617f.exe	cmp dword ptr [rbp+0x10], 0x4	memval[0x10ffb70]=0x10ffb70
		→ 0x4, immval = 0x04	
19 0x0ceca21	0d0617f.exe	jmp 0xceca25	

```

20 0x0ceca25 | 0d0617f.exe | jnz 0xceca41 |
21 0x0ceca27 | 0d0617f.exe | jmp 0xceca2e |
22 0x0ceca2e | 0d0617f.exe | cmp dword ptr [rbp+0x14], 0x0 | memval[0x10ffb74]=0x10ffb74
    -> 0x1, immval = 0x00
23 0x0ceca32 | 0d0617f.exe | jmp 0xceca35 |
24 0x0ceca35 | 0d0617f.exe | jz 0xcec3a4 |
25 0x0ceca3b | 0d0617f.exe | jmp 0xceca41 |

```

Os registradores **rcx** (linha 1), **rdx** (linha 2), **r8** (linha 3) e **r9** (linha 4) recebem, respectivamente, o *process handle*, a classe de informação (*ProcessDebugFlags*), o endereço para armazenar o valor retornado e o tamanho esperado (4 bytes). Em seguida, há um salto na linha 9 (**jmp r10**) para a função *ZwQueryInformationProcess*, interna ao *ntdll.dll*.

Após a função retornar, o código **test eax, eax** (linha 14) indica se **eax** é 0 (*STATUS_SUCCESS*) ou outro valor, desviando para diferentes blocos conforme o resultado.

O valor em **dword ptr [rbp+0x10]** (linha 18) que aponta ao endereço **0x10ffb70** verifica se foram efetivamente retornados 4 bytes (**0x4**), de modo a garantir que *NtQueryInformationProcess* tenha fornecido as informações esperadas. O valor em **dword ptr [rbp+0x14]** (linha 22) que aponta ao endereço **0x10ffb74** guarda o *ProcessDebugFlags* retornado. Se for **0x0** indica que há um debugger anexado ao processo atual (*NoDebugInherit* invertido). Caso contrário, significa ausência de depurador.

Ao consultar *ProcessDebugFlags*, o *Obsidium* verifica se há depuradores anexados ao processo atual, reforçando seu mecanismo anti-debug. Se o resultado for 0 (*debugger detectado*), o protetor pode tomar ações evasivas, como encerrar a execução ou usar outros métodos de defesa para dificultar a análise.

7.1.8 Técnica Anti-debug - Detecção de Depuradores com a Função *CheckRemoteDebuggerPresent*

A *Contradef* com o módulo *FunctionInterceptor* identificou a invocação à função *CheckRemoteDebuggerPresent*, o que sugere que o *Obsidium* também utiliza uma função específica para detectar a presença de um depurador, o que é uma técnica comum de evasão. A função é usada para verificar se o processo atual ou outro processo no sistema está sendo depurado, o que pode indicar a presença de ferramentas de análise ou depuração que o software deseja evitar.

Listagem 7.17: Invocações da função *CheckRemoteDebuggerPresent* para detecção de depuradores

```

1 [+] CheckRemoteDebuggerPresent ...
2   Module name: C:\Windows\System32\KERNELBASE.dll
3   Thread: 0
4   Call Id: 1
5   Routine address: 7ff800b50960
6   Parameters:
7       hProcess: 0
8       pbDebuggerPresent: FALSE
9   Return Value: 0
10 [*] Completed

```

No resultado da *Contradef*, a função *CheckRemoteDebuggerPresent* foi chamada ao menos 3 vezes com o parâmetro *hProcess* definido como 0, o que geralmente representa o processo atual, e *pbDebuggerPresent* retornou *FALSE*. Isso indica que essa função não detectou a presença de um depurador durante a execução com a *Contradef*.

Ferramentas de proteção como o *Obsidium* utilizam diferentes métodos de verificação para detectar depuradores ou ambientes de análise, visando aumentar suas chances de evasão. Métodos simples, como *CheckRemoteDebuggerPresent*, são capazes de identificar depuradores ocultos. O uso dessa função permite detectar uma variedade maior de depuradores e ambientes de análise,

garantindo uma cobertura mais abrangente. Esses diferentes métodos produzem resultados variados, pois cada técnica é adaptada para detectar ameaças em diferentes níveis e configurações do sistema.

7.2 Técnicas Evasivas Encontradas no *Themida*

Conforme dados consultados no *VirusTotal*, a amostra `a1c687574b51...`³ é um executável malicioso para o sistema operacional Windows de 64 bits, e identificado como um sofisticado *infostealer* com características das famílias *Redline Stealer* e *RisePro*. Para enganar a vítima, o *malware* se disfarça com o nome de arquivo *xml_magik.exe* e utiliza uma assinatura digital inválida que tenta se passar por um componente de *software* ERP legítimo (“WAPRO Fakir”). Uma vez executado, ele demonstra um comportamento agressivo de evasão e persistência, incluindo a desativação de funcionalidades do *Windows Defender* através de alterações no registro e WMI, a criação de tarefas agendadas para garantir sua execução contínua, a busca por processos de antivírus e o carregamento de extensões maliciosas em navegadores baseados em *Chromium* com o objetivo de exfiltrar dados sensíveis. Primeira aparição (primeira vez enviado para análise) em 30/04/2024 e última aparição (última submissão) em 08/05/2024.

7.2.1 Técnica anti-VM: inspeção do SMBIOS via `GetSystemFirmwareTable` (RSMB)

Esta técnica explora a API do Windows `GetSystemFirmwareTable` para obter, em *buffer*, as tabelas de firmware/BIOS contendo identificadores do fabricante, produto e plataforma. A tabela de firmware do sistema permite inferir se estão dentro de uma máquina virtual. A lógica por trás dessa técnica é que o SMBIOS — padrão mantido pela DMTF para expor metadados de *hardware/firmware* a *software* de gerenciamento ([Distributed Management Task Force \(DMTF\), 2025](#)) — provê um formato estável e amplamente implementado para descrever a plataforma. Logo, virtualizadores costumam preencher esses campos com identificadores próprios, o que viabiliza detecção com baixo custo e poucos falsos negativos.

Binários protegidos pelo *Obsidium* consultam esse *buffer* em busca de padrões de texto que denunciam virtualização (p.,ex., “VirtualBox”, “VMware, Inc.”, “innotek GmbH”, “Virtual Machine”, “Hyper-V” ou datas típicas de BIOS virtualizada), acionando desvios anti-análise quando detectados.

Para verificar o SMBIOS de um sistema, é possível chamar a API `GetSystemFirmwareTable`. Para verificar o SMBIOS, o primeiro parâmetro (`FirmwareTableProviderSignature`) deve ser RSMB. As informações do SMBIOS são armazenadas no *buffer* apontado pelo terceiro parâmetro (`pFirmwareTableBuffer`) ([Kim et al., 2022](#)).

A Listagem 7.18 (capturada pelo *FunctionInterceptor*) mostra a chamada à `GetSystemFirmwareTable` com o *provider* `0x52534d42` (ASCII ‘RSMB’, linha 6), *BufferSize* de 464 bytes (linha 9) e, sobretudo, as substituições aplicadas pela contramedida da *Contradef* no *payload* retornado (linhas 13–17). Em particular, os marcadores “innotek GmbH” e “VirtualBox” são substituídos por “PhoenixBIOS ” e “Dell BIOS ” (linhas 13–14), a data “12/01/2006” é trocada por “06/23/2017” (linha 15) e “Virtual Machine” por “Workstation ” (linha 16). Todas as substituições preservam o *comprimento* original para manter *offsets* e comparações de tamanho inalterados, minimizando a chance de detecção por checagens de integridade.

Listagem 7.18: Intercepção de `GetSystemFirmwareTable`(RSMB) e sanitização do SMBIOS

³<https://www.virustotal.com/gui/file/a1c687574b512e60c254447937836a11aca2ef11a928cf28c5e9e9138f7ce934>

```

1 [+] GetSystemFirmwareTable...
2   Thread: 0
3   Call Id: 0
4   Routine address: 0x7ff800b0ebe0
5   Parameters:
6       FirmwareTableProviderSignature: 0x52534d42
7       FirmwareTableID: 0x0
8       pFirmwareTableBuffer: 0x4cde90
9       BufferSize: 464 bytes
10  Address of calling function: 0x140747bfe
11 [-] Start of call GetSystemFirmwareTable
12  Returned from GetSystemFirmwareTable: 464 bytes
13  [Countermeasure/RSMB] "innotek GmbH" -> "PhoenixBIOS " (kept len 12) x3
14  [Countermeasure/RSMB] "VirtualBox" -> "Dell BIOS " (kept len 10) x3
15  [Countermeasure/RSMB] "12/01/2006" -> "06/23/2017" (kept len 10) x1
16  [Countermeasure/RSMB] "Virtual Machine" -> "Workstation " (kept len 15)
    x1
17  [Countermeasure/RSMB] Total patches: 8
18  Data from FirmwareTableBuffer (64 bytes): 0 2 5 0 c8 1 0 0 0 14 0 0 1 2 0 e0
    c 3 1 90 80 1 48 0 0 0 0 1 0 50 68 6f 65 6e 69 78 42 49 4f 53 20 0 44 65 6c 6
    c 20 42 49 4f 53 20 0 30 36 2f 32 33 2f 32 30 31 37 0 0
19  Operation completed successfully.
20 [-] Call from GetSystemFirmwareTable completed
21 [*] Completed

```

A Listagem 7.19 (trecho do `TraceInstructions`) evidencia como o binário consulta o *buffer* sanitizado: na **linha 1**, R15 aponta para "PhoenixBIOS ", resultado direto da contramedida (cf. List. 7.18, linha 13). Na **linha 2**, a instrução `cmp byte ptr [r15], r10b` compara o primeiro caractere do identificador lido ('P' = 0x50) com o esperado pelo protetor ('V' = 0x56), que é o primeiro caractere de um dos padrões procurados ("VirtualBox", "VMware, Inc." ou "Virtual Machine"); a comparação falha, evitando a detecção. Em seguida, o código não usa um *branch* condicional explícito (JZ/JNZ); em vez disso, `pushfq/pop` (linhas 3 e 5) capturam *flags* (incluindo *ZF*) e os serializam em memória (**linha 6**), constituindo um *estado* composto por várias operações aritméticas que definem o endereço para o salto incondicional (**linha 10**, `jmp r14`). Essa forma de ofuscação introduzida pelo mecanismo de virtualização de código mascara a dependência direta do resultado do `cmp`, dificultando heurísticas que procuram JZ/JNZ proximamente ao teste.

Listagem 7.19: Consulta ao SMBIOS sanitizado e desvio ofuscado (*branchless*)

```

1 0x1404cf568 | a1c6875.exe | mov r15, qword ptr [r15] | r15 = 0x4cdeac -> 0
    x4278696e656f6850 (4789694126577641552) -> "PhoenixBIOS "
2 0x1404cf56b | a1c6875.exe | cmp byte ptr [r15], r10b | memval [0x4cdeac] -> 0x50
    -> "PhoenixBIOS ", r10b = 0x56 -> "'V'"
3 0x1404cf56e | a1c6875.exe | pushfq | rsp = 0x14ff00
4 ... | ... | ... |
5 0x1404cf5b2 | a1c6875.exe | pop rdx | rdx = 0x297, rsp = 0
    x14ff08
6 0x1404cf5b3 | a1c6875.exe | mov qword ptr [r15], rdx | memval [0x1403d54f6] = 0
    x297
7 ... | ... | ... |
8 0x1404c465b | a1c6875.exe | jmp r14 |

```

Combinando `FunctionInterceptor` e `TraceInstructions`, demonstrou-se que (i) a detecção baseia-se em padrões de texto no SMBIOS (List. 7.18, linhas 13–16); (ii) o uso de ofuscação para evitar o uso de saltos condicionais dependentes da *ZF* definida na comparação (List. 7.19, linha 2); e (iii) a sanitização RSMB da *ContraDef* quebra a correspondência de padrões ('P' ≠ 'V' na linha 2 da List. 7.19) sem alterar o formato global do *buffer*, neutralizando a técnica anti-VM com impacto mínimo no fluxo legítimo.

7.2.2 Técnica anti-VM: consulta a chaves do Registro e artefatos ACPI/BIOS

Nesta técnica, o binário inspeciona chaves e valores do Registro do Windows em busca de indícios de virtualização e de execução em ambientes instrumentados. Esse método foi descrito por [Lee et al. \(2021\)](#), que também propuseram uma contramedida baseada na interceptação das funções `memmove` e `RegOpenKeyA`. Na *Contradef*, reimplementamos a contramedida por uma via distinta e mais aderente à diversidade de versões dos protetores comerciais: em vez de depender da ocorrência de `memmove`, como sugerido por [Lee et al. \(2021\)](#), realizamos a substituição de caracteres nas *strings* alvo *antes* das chamadas a API do Registro (`RegOpenKeyA/RegQueryValueExA`), restaurando-as *após* o retorno. Em termos práticos, nomes e identificadores (“VBOX”, “VirtualBox”, “Oracle VM”) são normalizados para alternativas benignas de mesmo comprimento (p.ex., “DELL”, “Dell BIOS”), de modo a preservar tamanhos de *buffer* e evitar efeitos colaterais na lógica de chamada. Essa abordagem é robusta frente a variações de implementação, pois algumas versões do *Themida* não utilizam `memmove` no caminho crítico, ao passo que a nossa estratégia, por atuar no ponto de consulta (API do Registro), permanece efetiva independentemente da função de cópia empregada pelo protetor.

Em termos práticos, observamos a abertura de chaves sob ramos de configuração do sistema e de *hardware*, seguida da leitura de valores associados a BIOS/firmware e a tabelas ACPI. De forma sucinta, ACPI (*Advanced Configuration and Power Interface*) é um padrão de firmware que define como o sistema operacional descobre e configura componentes de *hardware* (incluindo tabelas como a DSDT) ([UEFI Forum, Inc., 2021](#)), e é frequentemente utilizado por analisadores para identificar marcas de virtualização.

A Listagem 7.20 apresenta um resumo do trecho do log capturado pelo `FunctionInterceptor`, que mostra o padrão de consultas ao registro do Windows: (i) consulta a classes de dispositivos (adaptadores de vídeo) e a descrições de BIOS do sistema/vídeo (linhas 1–23); e (ii) enumeração de chaves ACPI sob `HARDWARE\ACPI\DSDT\VBOX__` (linhas 25–28), tradicionalmente associadas a ambientes VirtualBox. Conjuntamente, essas consultas revelam uma rotina que cruza metadados de inicialização do sistema, descrições de BIOS e a presença de tabelas ACPI com identificadores de fornecedor vinculados a hipervisores.

Listagem 7.20: Consultas ao Registro para detecção de virtualização (resumo dos logs capturados no `FunctionInterceptor`)

```

1 [+] RegOpenKeyA...
2 lpSubKey: SYSTEM\ControlSet001\Control\Class{4D36E968-E325-11CE-BFC1-08002
  BE10318}\0000
3 Return Value: 0
4 [+] RegQueryValueExA...
5 lpValueName: DriverDesc
6 Return Value: 0
7 [Countermeasure/REG] "VirtualBox" -> "Dell BIOS " (kept len 10) x1
8 ...
9 [+] RegOpenKeyA...
10 lpSubKey: Hardware\description\System
11 Return Value: 0
12 [+] RegQueryValueExA...
13 lpValueName: SystemBiosVersion
14 Return Value: 0
15 [Countermeasure/REG] "VBOX" -> "DELL" (kept len 4) x1
16 ...
17 [+] RegQueryValueExA...
18 lpValueName: VideoBiosVersion
19 Return Value: 0
20 [Countermeasure/REG] "VirtualBox" -> "Dell BIOS " (kept len 10) x1

```

```

21 [Countermeasure/REG] "Oracle VM" -> "Dell BIOS" (kept len 9) x1
22 ... (outras ocorrências) ...
23 [Countermeasure/REG] Total patches: 8
24 ...
25 [+] RegOpenKeyA...
26 lpSubKey: HARDWARE\ACPI\DSDT\VBOX__
27 Return Value: 2
28 [Countermeasure] Patched last char of lpSubKey

```

As leituras de `SystemBiosVersion` e `VideoBiosVersion` sob `Hardware\description\System` iniciam com a abertura da chave (linhas 9–11) e incluem as consultas/retornos (linhas 12–21), que costumam expor *strings* de fornecedor (“VirtualBox”, “VBOX”, “Oracle VM”) típicas de BIOS/firmware virtualizados. Já o ramo `HARDWARE\ACPI\DSDT\VBOX__` (linhas 25–28) referencia objetos ACPI cujos identificadores podem denunciar o hipervisor. Em conjunto, essas consultas fornecem ao protetor um critério robusto de *fingerprinting* do ambiente.

A técnica observada combina inspeções a chaves de controle do sistema, descrições de BIOS e publicações de tabelas ACPI no Registro para compor um veredito de “ambiente virtual”. Por apoiar-se em artefatos padronizados e facilmente consultáveis, ela é barata e efetiva — e, como salientado por Lee et al. (2021), é amplamente adotada por protetores comerciais.

7.2.3 Técnica anti-análise: verificação de nomes/títulos de janelas via FindWindowA

A técnica consiste em consultar diretamente a lista de janelas do sistema para inferir a presença de ferramentas de monitoramento pelo *nome da classe* da janela (`lpClassName`) ou pelo *título* (`lpWindowName`). A API do Windows, `FindWindowA/W`, retorna um `HWND` válido quando encontra uma janela de nível superior que corresponda a um dos parâmetros; caso contrário, retorna `NULL` (Microsoft Learn, 2024b). Na amostra, observou-se a sondagem explícita de utilitários da Sysinternals⁴ — versões antigas (*Regmon*, *Filemon*) e o *Process Monitor* moderno que é uma ferramenta de monitoramento avançada para *Windows* que mostra o sistema de arquivos em tempo real, o Registro e a atividade do processo/thread (Microsoft Sysinternals, 2025) — por meio de seus nomes de classe e/ou títulos característicos (`RegmonClass`, `FilemonClass`, `PROCMON_WINDOW_CLASS`, “*File/Process Monitor - Sysinternals: www.sysinternals.com*”).

A Listagem 7.21 (módulo `FunctionInterceptor`) evidencia sete invocações consecutivas a `FindWindowA`. Os parâmetros consultados foram, em sequência: `lpClassName = "RegmonClass"` (linha 4); `lpClassName = "Registry Monitor - Sysinternals: www.sysinternals.com"` (linha 12); `lpClassName = "18467-41"` (linha 20); `lpClassName = "FilemonClass"` (linha 28); `lpWindowName = "File Monitor - Sysinternals: www.sysinternals.com"` (linha 37); `lpClassName = "PROCMON_WINDOW_CLASS"` (linha 44); e `lpWindowName = "Process Monitor - Sysinternals: www.sysinternals.com"` (linha 53). Em todas as consultas, o retorno foi `0` (linhas 6, 14, 22, 30, 38, 46 e 54), indicando que nenhuma janela correspondente estava presente no momento da verificação.

Listagem 7.21: Sondagem de janelas por classe/título com `FindWindowA` (`FunctionInterceptor`)

```

1 [+] FindWindowA...
2   Module name: C:\Windows\System32\USER32.dll
3   Parameters:
4       lpClassName: RegmonClass
5       lpWindowName:
6   Return Value: 0
7 [*] Completed

```

⁴<https://learn.microsoft.com/en-us/sysinternals/>

```

8
9 [+] FindWindowA...
10 Module name: C:\Windows\System32\USER32.dll
11 Parameters:
12     lpClassName: Registry Monitor - Sysinternals: www.sysinternals.com
13     lpWindowName:
14 Return Value: 0
15 [*] Completed
16
17 [+] FindWindowA...
18 Module name: C:\Windows\System32\USER32.dll
19 Parameters:
20     lpClassName: 18467-41
21     lpWindowName:
22 Return Value: 0
23 [*] Completed
24
25 [+] FindWindowA...
26 Module name: C:\Windows\System32\USER32.dll
27 Parameters:
28     lpClassName: FilemonClass
29     lpWindowName:
30 Return Value: 0
31 [*] Completed
32
33 [+] FindWindowA...
34 Module name: C:\Windows\System32\USER32.dll
35 Parameters:
36     lpClassName:
37     lpWindowName: File Monitor - Sysinternals: www.sysinternals.com
38 Return Value: 0
39 [*] Completed
40
41 [+] FindWindowA...
42 Module name: C:\Windows\System32\USER32.dll
43 Parameters:
44     lpClassName: PROCMON_WINDOW_CLASS
45     lpWindowName:
46 Return Value: 0
47 [*] Completed
48
49 [+] FindWindowA...
50 Module name: C:\Windows\System32\USER32.dll
51 Parameters:
52     lpClassName:
53     lpWindowName: Process Monitor - Sysinternals: www.sysinternals.com
54 Return Value: 0
55 [*] Completed

```

O `TraceFncCall` confirma o mesmo padrão em nível de chamadas (Listagem 7.22): as linhas 1–2 mostram a consulta por "FilemonClass" com retorno 0; as linhas 3–4, a verificação pelo título "File Monitor - Sysinternals: www.sysinternals.com"; as linhas 5–6, a sondagem por "PROCMON_WINDOW_CLASS"; e, por fim, as linhas 7–8, a checagem do título "Process Monitor - Sysinternals: www.sysinternals.com". Em todas as ocorrências, o valor de retorno foi 0, indicando ausência de HWND correspondente no ambiente no instante da consulta.

Listagem 7.22: Padrão de chamadas `FindWindowA` (`TraceFncCall`)

```

1 Call from 0x1404b242c | a1c6875.exe -> 0x7ff802bbaae0 | USER32.dll:FindWindowA
  (0x1405342dd -> "FilemonClass", 0, 0, 0, 0x14ff28, 0x206, ...)
2 Ret from 0x7ff802bbaaf7 | USER32.dll:FindWindowA+0x00017 Return value -> 0 (

```

```

    next 0x140743888, came-from 0)
3 Call from 0x1404b242c | a1c6875.exe -> 0x7ff802bbaae0 | USER32.dll:FindWindowA
  (0, 0x1405391ad -> "File Monitor - Sysinternals: www.sysinternals.com", 0
  x2f10e40, 0x1, 0x14ff28, 0x206, ...)
4 Ret from 0x7ff802bbaaf7 | USER32.dll:FindWindowA+0x00017 Return value -> 0 (
  next 0x140743cf1, came-from 0)
5 Call from 0x1404b242c | a1c6875.exe -> 0x7ff802bbaae0 | USER32.dll:FindWindowA
  (0x140539713 -> "PROCMON_WINDOW_CLASS", 0, 0x2f11440, 0x1, 0x14ff28, 0x206,
  ...)
6 Ret from 0x7ff802bbaaf7 | USER32.dll:FindWindowA+0x00017 Return value -> 0 (
  next 0x140744118, came-from 0)
7 Call from 0x1404b242c | a1c6875.exe -> 0x7ff802bbaae0 | USER32.dll:FindWindowA
  (0, 0x1405395a2 -> "Process Monitor - Sysinternals: www.sysinternals.com", 0
  x2f11080, 0x1, 0x14ff28, 0x206, ...)
8 Ret from 0x7ff802bbaaf7 | USER32.dll:FindWindowA+0x00017 Return value -> 0 (
  next 0x140744532, came-from 0)

```

Em termos de *trade-offs*, trata-se de uma técnica de baixo custo (uma única chamada a USER32 por verificação), eficaz para identificar rapidamente a execução de ferramentas conhecidas, mas sujeita a *falsos negativos* (p.ex., quando as janelas estão ocultas/inexistentes) e *falsos positivos* (padrões de texto no título de softwares legítimos). Ainda assim, o padrão observado — alternar buscas por *class name* e *window title* — é típico de empacotadores e *protectors* comerciais que tentam sinalizar a presença de observabilidade no ambiente antes de prosseguir com estágios sensíveis da execução.

7.2.4 Técnica anti-*sandbox*: sondagem de DLLs via GetModuleHandleExW

Esta técnica consiste em consultar, por meio de `GetModuleHandleExW`, se módulos característicos de ambientes de isolamento (*sandboxes*) já estão mapeados no processo em análise. Pela API do Windows, `GetModuleHandleExW` que devolve um `BOOL` indicando sucesso; o `HMODULE` é retornado via parâmetro de saída `phModule`. Quando não há módulo carregado com o nome consultado, a função retorna `FALSE` e `*phModule` permanece `NULL` (Microsoft Learn, 2024c). Na amostra, os alvos foram `cmdvrt32.dll` (associada à virtualização do *Comodo Internet Security* (FreeFixer / SystemExplorer, 2024)) e `SbieDll.dll` (biblioteca principal de isolamento da *Sandboxie* (Sandboxie-Plus Project, 2024b,a,c)).

A Listagem 7.23 apresenta o log do `TraceFncCall.M2`. A primeira consulta chama `GetModuleHandleExW` com o nome "`cmdvrt32.dll`" (linha 1) e recebe retorno `0` (falha) na linha 2; a segunda repete o procedimento para "`SbieDll.dll`" (linha 4), novamente com retorno `0` (falha). Em ambos os casos, a falha no retorno (valor 0) indica que tais DLLs não estavam carregadas no contexto do processo no momento da verificação.

Listagem 7.23: Consultas a módulos de sandbox com `GetModuleHandleExW`

```

1 Call from 0x1405360ec | a1c6875.exe -> 0x7ff800b30785 | KERNELBASE.dll:
  GetModuleHandleExW (0x2, 0x4cd880 -> "cmdvrt32.dll", 0x14fee0)
2 Ret from 0x7ff800b3085b | KERNELBASE.dll:GetModuleHandleExW Return value -> 0 (
  next 0x140640b3c)
3 ...
4 Call from 0x00000001405360ec | a1c6875.exe -> 0x7ff800b30785 | KERNELBASE.dll:
  GetModuleHandleExW (0x2, 0x4c4e20 -> "SbieDll.dll", 0x14fee0)
5 Ret from 0x7ff800b3085b | KERNELBASE.dll:GetModuleHandleExW Return value -> 0 (
  next 0x140640b3c)

```

A detecção de *sandbox* é inferida a partir do conteúdo escrito no parâmetro de saída `phModule`. Na amostra, o mesmo ponteiro é reutilizado nas duas consultas a `GetModuleHandleExW` ("`cmdvrt32.dll`" e "`SbieDll.dll`"), conforme as linhas 1 e 4 da Listagem 7.23, nas quais `phModule = 0x14fee0`.

O trecho da Listagem 7.24, obtido com o `TraceInstructions`, mostra primeiro o retorno da API (linha 1, `GetModuleHandleExW | ret`) finalizando a segunda chamada à `GetModuleHandleExW` (consulta do módulo "SbieDll.dll"); após algumas instruções, o valor de retorno booleano é recuperado em `RAX` (`pop rax`, linha 3), indicando `FALSE` (0), e o endereço de saída `phModule` é restabelecido em registradores/pilha (`pop rdi`, linha 4, e realocação da pilha para `0x14fee0` nas linhas 6–10). A leitura de `[*phModule]` (linhas 12–15) obtém 0 em `RAX`, evidenciando que o `HMODULE` retornado é `NULL`; esse zero é, por fim, consumido no desvio `test/jz` (linhas 17–18). Em outras palavras, não há módulo mapeado para os nomes consultados e a verificação falha, como também indicado pelos *returns* 0 (`FALSE`) nas linhas 2 e 5 da Listagem 7.23.

Listagem 7.24: Pós-`GetModuleHandleA`: retorno do handle `NULL`

```

1 0x7ff800b3085b | GetModuleHandleExW | ret          | rsp = 0x14fe20          ;
    retorno da API
2 ...           | ...                 | ...                     |
3 0x140640bf1    | a1c6875.exe         | pop rax                 | rax = 0, rsp = 0x14fe88
    ; B00L=0
4 0x140640bf2    | a1c6875.exe         | pop rdi                 | rdi = 0x14fee0, rsp=0
    x14fe90 ; phModule = 0x14fee0
5 ...           | ...                 | ...                     |
6 0x140721f1c    | a1c6875.exe         | push rsp                |
7 0x140721f1d    | a1c6875.exe         | pop r11                 | r11 = 0x14fee0
    ; captura phModule em r11
8 ...           | ...                 | ...                     |
9 0x140721f2a    | a1c6875.exe         | xchg [rsp], r11         | r11 = 0x14fda0
10 0x140721f2e   | a1c6875.exe         | mov  rsp, [rsp]         | rsp = 0x14fee0
    ; pilha muda p/ phModule
11 ...          | ...                 | ...                     |
12 0x140721fa0   | a1c6875.exe         | push [rsp]              | ; (valor em *phModule) = 0
13 0x140721fa3   | a1c6875.exe         | push [rsp]              | ; (duplica o valor 0)
14 ...          | ...                 | ...                     |
15 0x140721fbc   | a1c6875.exe         | mov  rax,[rsp]          | rax = 0x0    ; *phModule
    == NULL
16 ...          | ...                 | ...                     |
17 0x140640c9a   | a1c6875.exe         | test eax,eax            | eax = 0
18 0x140640c9c   | a1c6875.exe         | jz  0x140531875         |

```

A ofuscação do *Themida* dissocia a causa (leitura de `*phModule` após a chamada de API) do efeito (bifurcação “detectado/não detectado”), distribuindo essa lógica em blocos curtos com operações de pilha e aritmética (`push/pop/add/sub`) e saltos indiretos. Essa dispersão quebra a localidade semântica e dificulta relacionar imediatamente o resultado da API com a decisão final, elevando o custo da análise. Ainda assim, a ausência de `HMODULE` nas duas consultas (`Return value -> 0`, Listagem 7.23, linhas 2 e 5) é consistente com um ambiente sem *sandbox* detectável no momento da verificação. Além disso, a *Contradef* capturou valores (incluindo texto) dos registradores e dos endereços de memória, o que facilitou a interpretação da lógica de execução, mesmo com os métodos de ofuscação empregados pelo *Themida*.

7.2.5 Técnica anti-análise: enumeração de módulos do sistema via `RtlGetNativeSystemInformation` com varredura de padrões de texto

Esta técnica consulta a lista global de módulos do sistema (*drivers* em modo kernel) para detectar artefatos de ferramentas de monitoramento/depuração. O *Themida* enumera *drivers* em modo kernel consultando a classe `SystemModuleInformation` por meio da rotina nativa `RtlGetNativeSystemInformation`. Embora essa função não seja oficialmente documentada, seu

uso é análogo a `NtQuerySystemInformation`, que recupera informações do sistema de acordo com o valor definido em `SystemInformationClass` (primeiro parâmetro) (DeceptiveBytes, 2018; Microsoft Learn, 2025). A consulta é feita com `SystemInformationClass = 11 (0xb)`, que corresponde a `SystemModuleInformation` e retorna a lista de módulos do sistema (Joseph Landry, 2016). Em seguida, para cada módulo retornado, o código (i) valida progressivamente o ponteiro do nome com chamadas repetidas a `IsBadReadPtr`; (ii) “encurta” até isolar o nome base (p.ex., `\SystemRoot\System32\kd.dll` → `kd.dll`); e (iii) compara o nome isolado, de forma *case-insensitive*, contra padrões de texto (assinatura) associados a ferramentas de monitoramento/forense conhecidas. Esse processo se repete para cada assinatura, como REGMON⁵, Filemon⁶ e PROCMON⁷.

A Listagem 7.25 (`FunctionInterceptor`) mostra a chamada a `RtlGetNativeSystemInformation` com `SystemInformationClass = 11` (linha 3), o endereço do *buffer* de saída em `SystemInformation = 0x229c0000` (linha 4) e o tamanho máximo de 65536 bytes (linha 5). A chamada retorna sucesso (`Return Value: 0`, linha 8).

Listagem 7.25: Interceptação e reescrita de `SYSTEM_MODULE_INFORMATION`

```

1 [+] RtlGetNativeSystemInformation...
2   Parameters:
3       SystemInformationClass: 11 (0xb)
4       SystemInformation: 229c0000
5       SystemInformationLength: 65536 bytes
6       ReturnLength (Address): 0
7 [-] Start of obtaining system information
8   Return Value: 0
9 [-] Obtaining system information completed
10 [-] Decoding / rewriting SYSTEM_MODULE_INFORMATION -> one entry (kd.dll)
11   Original modules: 157
12   Rewritten to 1 module: kd.dll
13 [-] SYSTEM_MODULE_INFORMATION rewrite completed
14 [*] Completed

```

A *Contradef* aplicou, então, a **reescrita** do conjunto `SYSTEM_MODULE_INFORMATION` observado na Listagem 7.25 (linhas 10–13), reduzindo **157 entradas** a apenas **uma** (`kd.dll`, linha 12). Essa contramedida tem dois efeitos imediatos: (i) impede que a amostra encontre módulos indicativos de ferramentas de análise e (ii) *colapsa* a iteração subsequente, reduzindo drasticamente o número de operações (menos chamadas `IsBadReadPtr/lstrcmpiA`) e, por consequência, o volume de eventos registrados.

A Listagem 7.26 ilustra o padrão de varredura capturado com o `TraceFcnCall`: uma chamada a `RtlGetNativeSystemInformation` (linhas 1–2) seguida, para **cada módulo**, de uma sequência de `IsBadReadPtr` que desloca o início da sequência de texto até restar o nome curto (linhas 4–14) e, então, a comparação `lstrcmpiA` com uma assinatura (linha 15). Esse bloco (*validação* → *comparação*) repete-se **para cada um dos seguintes padrões**: REGMON, regsys, sysregm, PROCMON, Kernel Detective, CisUtMonitor, Revoflt, FileMonitor.sys e Filem.

Listagem 7.26: Padrão de varredura: validação e comparação de nomes de módulos

```

1 Call from 0x1404b242c | a1c6875.exe -> 0x7ff8032adba0 | ntdll.dll:
   RtlGetNativeSystemInformation (0xb, 0x229c0000, 0x10000, 0, 0x14fb60, 0x202,
   ...)
2 Ret from 0x7ff8032adbb4 | ntdll.dll:RtlGetNativeSystemInformation Return value
   -> 0 (next 0x1406a6f26, came-from 0)
3

```

⁵<https://learn.microsoft.com/sysinternals/downloads/regmon/>

⁶<https://learn.microsoft.com/sysinternals/downloads/filemon/>

⁷<https://learn.microsoft.com/sysinternals/downloads/procmom/>

```

4 Para cada modulo executa:
5
6 Call from 0x1404b242c | a1c6875.exe -> 0x7ff802935de0 | KERNEL32.DLL:
  IsBadReadPtr (0x229c0030 -> "\SystemRoot\System32\kd.dll", 0x4, 0x14fec8, 0
  x140293000, 0x246 -> "F", 0x202, ...)
7 Ret from 0x7ff802935e5a | KERNEL32.DLL:IsBadReadPtr Return value -> 0 (next 0
  x1406a7083, came-from 0)
8 Call from 0x1404b242c | a1c6875.exe -> 0x7ff802935de0 | KERNEL32.DLL:
  IsBadReadPtr (0x229c0031 -> "SystemRoot\System32\kd.dll", 0x4, 0x229c0030 ->
  "\SystemRoot\System32\kd.dll", 0x1000, 0x14fef8 -> "1", 0x202, ...)
9 Ret from 0x7ff802935e5a | KERNEL32.DLL:IsBadReadPtr Return value -> 0 (next 0
  x1406a7213, came-from 0)
10 Call from 0x1404b242c | a1c6875.exe -> 0x7ff802935de0 | KERNEL32.DLL:
  IsBadReadPtr (0x229c0032 -> "ystemRoot\System32\kd.dll", 0x4, 0x229c0031 ->
  "SystemRoot\System32\kd.dll", 0x1000, 0x14fef8 -> "2", 0x202, ...)
11 Ret from 0x7ff802935e5a | KERNEL32.DLL:IsBadReadPtr Return value -> 0 (next 0
  x1406a7213, came-from 0)
12 ... IsBadReadPtr se repete ate chegar a "kd.dll" (somente nome)
13 Call from 0x1404b242c | a1c6875.exe -> 0x7ff802935de0 | KERNEL32.DLL:
  IsBadReadPtr (0x229c0045 -> "kd.dll", 0x1e, 0x229c004b, 0x1000, 0x14fef8 ->
  "E", 0x202, ...)
14 Ret from 0x7ff802935e5a | KERNEL32.DLL:IsBadReadPtr Return value -> 0 (next 0
  x1406a74bc, came-from 0)
15 Call from 0x1404b242c | a1c6875.exe -> 0x7ff80293b640 | KERNEL32.DLL:lstrcmpiA
  (0x14053907a -> "REGMON", 0x229c0045 -> "kd.dll", 0x229c0045 -> "kd.dll", 0
  x1000, 0x282, 0x202, ...)

```

A técnica combina enumeração nativa de módulos com comparações de texto para inferir a presença de ferramentas de análise. A reescrita do `SYSTEM_MODULE_INFORMATION` pela *Contradef* (Listagem 7.25) neutraliza a detecção por artefatos e, ao mesmo tempo, reduz a complexidade operacional da varredura, oferecendo uma linha de defesa prática contra esse padrão de anti-análise. Além disso, a capacidade da *Contradef* de inspecionar parâmetros de chamada, conteúdos referenciados por ponteiros e regiões de memória alcançadas pelos endereços instrumentados permitiu capturar padrões textuais (*strings*) característicos de sondagens do ambiente. Com isso, foi possível identificar buscas explícitas por ferramentas específicas de análise — evidenciadas por nomes de módulos — a partir dessas assinaturas.

7.2.6 Outras técnicas anti-debug observadas no *Themida*

As técnicas a seguir foram identificadas no *Themida* e empregam os mesmos princípios gerais já discutidos anteriormente com a amostra do *Obsidium*: invocar funções da API do *Windows* para aferir sinais de depuração e, a partir disso, ajustar o fluxo de execução. A descrição abaixo é baseada nos logs coletados pelo *FunctionInterceptor* para a amostra do *Themida*.

7.2.6.1 Detecção via *NtQueryInformationProcess/ProcessDebugPort*

A amostra consulta o valor de `ProcessDebugPort` (código 7 em `PROCESSINFOCLASS`) por meio de `NtQueryInformationProcess`, técnica clássica para inferir a presença de depurador a partir do *debug port* associado ao processo (Microsoft Learn, 2023b; Check Point Research, 2022a). A Listagem 7.27 mostra a chamada com `ProcessInformationClass = 7` (linha 7), o *pseudohandle* do processo atual em `ProcessHandle = 0xffffffffffffffff` (linha 6), o *buffer* de saída `ProcessInformation = 0x14ff00` (linha 8) e o comprimento de 8 bytes (linha 9). A operação retorna `STATUS_SUCCESS` (linha 13).

Listagem 7.27: `NtQueryInformationProcess` com `ProcessDebugPort`

```

1 [+] NtQueryInformationProcess...
2   Thread: 0
3   Call Id: 0
4   Routine address: 0x7ff8032ad800
5   Parameters:
6       ProcessHandle: 0xffffffffffffffff
7       ProcessInformationClass: 7
8       ProcessInformation: 0x14ff00
9       ProcessInformationLength: 8
10      ReturnLength: 0x0
11   Address of calling function: 0x140709d98
12 [-] Start of call NtQueryInformationProcess
13   Returned from NtQueryInformationProcess (NTSTATUS): 0x0
14   Successful operation (STATUS_SUCCESS).
15 [-] Call from NtQueryInformationProcess completed
16 [*] Completed

```

De forma geral, quando o *debug port* está ausente, o valor armazenado em *ProcessInformation* é zero; caso contrário, um valor não nulo pode indicar depurador anexado ([Check Point Research, 2022a](#)).

7.2.6.2 Detecção via *NtQueryInformationProcess/ProcessDebugObjectHandle*

A amostra também consulta *ProcessDebugObjectHandle* (código 30, 0x1e) por *NtQueryInformationProcess*, buscando obter um *HANDLE* para o *debug object* associado ao processo ([Check Point Research, 2022a](#)). Na Listagem 7.28, a chamada usa novamente o *pseudohandle* (linha 6), define *ProcessInformationClass* = 30 (linha 7), aponta o mesmo *buffer* de saída 0x14ff00 (linha 8) e 8 bytes (linha 9). O retorno é um *NTSTATUS* de falha 0xc0000353 (linha 13), indicando que nenhum *debug object* válido foi obtido.

Listagem 7.28: *NtQueryInformationProcess* com *ProcessDebugObjectHandle*

```

1 [+] NtQueryInformationProcess...
2   Thread: 0
3   Call Id: 2
4   Routine address: 0x7ff8032ad800
5   Parameters:
6       ProcessHandle: 0xffffffffffffffff
7       ProcessInformationClass: 30
8       ProcessInformation: 0x14ff00
9       ProcessInformationLength: 8
10      ReturnLength: 0x0
11   Address of calling function: 0x140709efc
12 [-] Start of call NtQueryInformationProcess
13   Returned from NtQueryInformationProcess (NTSTATUS): 0xc0000353
14   Operation failed (NTSTATUS != 0).
15 [-] Call from NtQueryInformationProcess completed
16 [*] Completed

```

7.2.6.3 Detecção via *CheckRemoteDebuggerPresent*

Por fim, a amostra consulta *CheckRemoteDebuggerPresent*, função da API do *Windows* que retorna um *BOOL* indicando sucesso/fracasso da chamada e, via parâmetro de saída, se há depurador presente ([Microsoft Learn, 2024a](#); [Sikorski and Honig, 2012](#)). Na Listagem 7.29, o parâmetro *hProcess* é passado com o valor exibido no log (linha 7), e *pbDebuggerPresent* veio como *FALSE* (linha 8). O retorno da função foi 1 (linha 9), sinalizando chamada bem-sucedida e ausência de depurador no momento da verificação.

Listagem 7.29: `CheckRemoteDebuggerPresent` no processo corrente

```
1 [+] CheckRemoteDebuggerPresent...
2   Module name: C:\Windows\System32\KERNELBASE.dll
3   Thread: 0
4   Call Id: 0
5   Routine address: 7ff800b50960
6   Parameters:
7       hProcess: 18446744073709551615
8       pbDebuggerPresent: FALSE
9   Return Value: 1
10 [*] Completed
```

O *Themida* combina consultas a `NtQueryInformationProcess` (para `ProcessDebugPort` e `ProcessDebugObjectHandle`) com a verificação de `CheckRemoteDebuggerPresent` para inferir, por múltiplos sinais, a presença de um depurador. Esses mecanismos são frequentes no ecossistema *Windows* (Microsoft Learn, 2023b, 2024a; Sikorski and Honig, 2012) e se alinham aos princípios já discutidos anteriormente: sondar funções da API do *Windows* para indicadores de depuração e decidir, a partir deles, como prosseguir a execução.

7.3 Resumo das Técnicas Evasivas Encontradas

Categoria	API/Artefato	Indicador / Descrição
Anti-debug	<code>EnumWindows</code>	Busca por padrões em títulos de janelas de depuradores; presença indica ambiente de análise.
Anti-análise	<code>GetTickCount</code>	Medição de latência/overhead; desvios sugerem VM/DBI ou instrumentação ativa.
Anti-debug	<code>RaiseException</code>	Teste de exceção para inferir depurador (manuseio anômalo do SEH sugere depuração).
Anti-debug	<code>CreateFileW</code>	Sondagem de artefatos do TitanHide (<i>driver</i> /serviço); sucesso sugere ocultação de depurador.
Anti-debug	<code>NtQueryInformationProcess</code> (<code>ProcessDebugPort</code>)	Valor de porta de depuração $\neq 0$ indica depurador anexado.
Anti-debug	<code>NtQueryInformationProcess</code> (<code>ProcessDebugObjectHandle</code>)	<i>Handle</i> válido para objeto de depuração aponta para contexto de depuração.
Anti-DBI/Dbg	<code>NtQueryInformationProcess</code> (<code>ProcessDebugFlags</code>)	Inconsistências/normalizações em <i>flags</i> podem evidenciar manipulação por instrumento/depurador.
Anti-debug	<code>CheckRemoteDebuggerPresent</code>	Verificação direta de depurador anexado (parâmetro de saída sinaliza presença).

Tabela 7.1: Resumo das técnicas evasivas observadas no *Obsidium*.

7.3.1 Discussão

Os resultados apresentados neste capítulo evidenciam que a análise de amostras protegidas por *Obsidium* e *Themida* depende não apenas da capacidade de registrar eventos em tempo de execução, mas também da aplicação de contramedidas capazes de manter a execução do binário sob instrumentação. Nesse sentido, as técnicas de contorno de anti-VM e anti-DBI descritas por Lee et al. (2021) foram decisivas para a continuidade da análise, pois esses protetores comerciais podem interromper a execução, alterar o fluxo ou ocultar o comportamento de interesse quando identificam indícios de ambiente virtualizado, depurador ou instrumentador dinâmico.

Categoria	API/Artefato	Indicador / Descrição
Anti-VM	GetSystemFirmwareTable (RSMB)	Leitura de SMBIOS/ACPI; cadeias de fabricante/produto típicas de VM denunciam virtualização.
Anti-VM	Registro/ACPI/BIOS	Consultas a chaves do Registro e objetos ACPI/BIOS (p. ex., <code>HARDWARE\ACPI\DSDT\VBOX__</code>); presença de identificadores de hipervisor.
Anti-análise	FindWindowA	Verificação de nomes/títulos de janelas de ferramentas de análise/depuração.
Anti-análise	GetModuleHandleExW	Sondagem de DLLs/artefatos de análise previamente carregados no processo.
Anti-análise	RtlGetNativeSystemInformation	Enumeração/varredura de módulos do sistema com busca de padrões textuais e artefatos característicos.
Anti-debug	NtQueryInformationProcess (ProcessDebugPort)	Porta de depuração $\neq 0$ sugere depurador anexado.
Anti-debug	NtQueryInformationProcess (ProcessDebugObjectHandle)	<i>Handle</i> de objeto de depuração presente indica contexto de depuração.
Anti-debug	CheckRemoteDebuggerPresent	Sinalização de depurador anexado via parâmetro de saída.

Tabela 7.2: Resumo das técnicas evasivas observadas no *Themida*.

Compreender como *malwares* e protetores comerciais evitam a detecção é fundamental para fortalecer estratégias de análise. Nesse sentido, os *insights* fornecidos pela *Contradef* mostraram-se valiosos para identificar padrões de comportamento e para embasar heurísticas de detecção. Ao expor, em nível de chamadas de API, memória e instruções, a interação do binário com o sistema, a *Contradef* permite uma leitura mais fina das técnicas de ocultação, contribuindo tanto para o aprimoramento de ferramentas quanto para pesquisas acadêmicas.

Os resultados obtidos evidenciam um conjunto de características evasivas recorrentes e, em muitos casos, redundantes. Essa redundância é intencional: técnicas que, isoladamente, seriam simples (p. ex., sondagem de janelas, checagem de chaves de registro, inspeção de módulos do sistema), quando combinadas e distribuídas pelo fluxo, elevam o custo da análise e reduzem a previsibilidade do ambiente de execução. No *Obsidium*, por exemplo, observou-se uma composição de verificações que se reforçam mutuamente, demandando contramedidas que atuem de forma ampla e coordenada para evitar que algum vetor remanescente acione a evasão.

No *Themida*, a dificuldade é ampliada pelos mecanismos de ofuscação e *virtualização de código*: as instruções originais são traduzidas para uma máquina virtual proprietária, e os registradores e *OPCODEs* variam entre *builds*. Assim, embora exista um “esqueleto” de VM comum, os detalhes (layout de registradores virtuais, tabela de *OPCODEs*) mudam, prejudicando a reutilização de assinaturas estáticas e de *scripts* de desvirtualização. Na prática, o analista precisa primeiro compreender o modelo comum e, só então, adaptar a análise às particularidades de cada instância protegida.

Outro aspecto prático observado é a variabilidade entre execuções: proteções que descompactam e realocam trechos de código podem alterar endereços, sequências e pontos de instrumentação a cada rodada. Com isso, a correspondência entre os dados dos módulos da *Contradef* (p. ex., *TraceFncCall*, *TraceMemory*, *TraceInstructions*) torna-se mais efetiva quando os artefatos são coletados na mesma execução. Ainda assim, a ferramenta oferece estratégias complementares em diferentes níveis de profundidade: o *TraceInstructions* fornece uma visão detalhada do fluxo efetivamente executado; o *FunctionInterceptor* revela parâmetros e *buffers* de saída para hipóteses sobre intenção; e as rotinas de reescrita/sanitização permitem testar contramedidas e validar hipóteses de detecção.

Assim, as técnicas anti-DBI e anti-VM não devem ser vistas apenas como obstáculos à

análise, mas também como elementos centrais para compreender a lógica de autoproteção dos protetores comerciais. Ao implementar e aprimorar contramedidas para essas técnicas, a *Contradef* conseguiu ultrapassar a fase inicial de detecção do ambiente e manter a execução instrumentada por tempo suficiente para revelar comportamentos posteriores. Esse resultado reforça que, em cenários envolvendo protetores como *Obsidium* e *Themida*, a análise dinâmica efetiva exige uma combinação entre observação, mitigação seletiva e correlação de evidências em múltiplos níveis.

Além disso, os *logs* produzidos pela *Contradef* constituem um artefato analítico relevante, pois preservam uma trilha verificável da execução monitorada. Para o analista, esses registros não servem apenas como evidência de que determinada API foi chamada, mas como base para reconstruir a sequência causal dos eventos: qual rotina foi executada, quais parâmetros foram utilizados, quais valores foram retornados, quais instruções interpretaram esses valores e como o fluxo de execução foi afetado. Dessa forma, a saída da ferramenta permite transformar eventos de baixo nível em indícios contextualizados de comportamento evasivo, favorecendo a validação manual, a identificação de comportamento evasivo, a comparação entre amostras e a criação de novas contramedidas.

De forma geral, a análise combinada (estática com o YARA + dinâmica com o Pin) mostrou-se essencial para identificar e, por vezes, neutralizar técnicas evasivas. A “força” dos protetores não decorre apenas da complexidade de cada técnica isolada, mas do acoplamento e da sobreposição entre elas. Isso reforça a necessidade de abordagens mais abrangentes e integradas — tanto para análise quanto para mitigação — capazes de acompanhar a evolução constante dos métodos de proteção. Em síntese, a experiência com *Obsidium* e *Themida* indica que defesas eficazes dependem de visibilidade multicamada, adaptabilidade e defesa (contramedidas). Essas diretrizes elevam a robustez das investigações e sustentam respostas mais rápidas e assertivas diante de *malwares* protegidos por ferramentas modernas.

Capítulo 8

Conclusões, Contribuições e Trabalhos Futuros

8.1 Considerações Finais

A investigação conduzida ao longo desta dissertação evidenciou que *malwares* protegidos por *packers* e protetores comerciais — em especial *Obsidium* e *Themida* — exploram um conjunto cada vez mais amplo e redundante de técnicas de evasão. Não se trata apenas das rotinas clássicas de *anti-debugging* e *anti-VM*, mas de cadeias completas de verificação que incluem consultas a estruturas internas do Windows (`ProcessDebugPort`, `ProcessDebugObjectHandle`, `ProcessDebugFlags`), inspeção de *drivers* de ocultação (como o *TitanHide*), uso de exceções estruturadas como canal de detecção, chamadas indiretas ou empilhadas para APIs sensíveis e virtualização de código. Esse encadeamento de verificações aumenta a resiliência do código malicioso e reduz a eficácia de ferramentas que adotam uma única estratégia de transparência.

A adoção de instrumentação binária dinâmica, com o *Intel Pin*, mostrou-se adequada para tornar visíveis essas etapas de proteção em tempo de execução, permitindo registrar **quando** o binário consulta o ambiente, **que** artefatos procura e **como** altera o fluxo a partir daí. Por outro lado, os experimentos também confirmaram o que a literatura recente tem apontado: o próprio DBI introduz artefatos detectáveis e *overhead* mensurável, de modo que qualquer solução realmente útil para análise de *malware* evasivo precisa combinar observação com normalização (contramedidas) — isto é, não apenas registrar o acesso às APIs sensíveis, mas também devolver respostas coerentes com uma execução nativa. Esse equilíbrio entre visibilidade e transparência foi um dos eixos centrais deste trabalho.

Nesse contexto, a ferramenta proposta, **Contradef**, foi concebida como um analisador dinâmico orientado a *comportamento evasivo* em executáveis Windows. Sua arquitetura buscou manter modularidade e seletividade, de modo a não instrumentar o binário inteiro de forma indiscriminada. Entre os recursos que se mostraram mais efetivos destacam-se:

- (i) o uso de regras *YARA* e de pontos de interesse previamente definidos, que permitem acionar apenas os módulos relevantes, reduzindo o volume de *logs* e o impacto de desempenho;
- (ii) a instrumentação condicional de memória e instruções, ativada somente diante de eventos específicos (p.,ex., chamada de API sensível, identificação de acesso ao PEB/TIB ou leitura de artefato de virtualização);

- (iii) a possibilidade de manipular parâmetros e valores de retorno, forçando caminhos de execução que o *malware* só trilharia sob certas condições ambientais;
- (iv) a capacidade de registrar o comportamento ou aplicar alterações dinâmicas sobre sequências identificadas com o Detector de Sequências de Instruções, o que aproxima a ferramenta de um modelo de observação ou manipulação de execução sob demanda.

Os experimentos indicaram que, mesmo diante de amostras empacotadas ou fortemente ofuscadas, a **Contradef** conseguiu expor rotinas críticas, extrair artefatos de memória e reconstruir trechos de fluxo de controle, validando o desenho de que primeiro se observa e só então se mitiga. Além disso, embora o foco desta dissertação tenha sido a análise de código malicioso, o mesmo conjunto de mecanismos mostrou-se aplicável a binários legítimos que empregam ofuscação, empacotamento ou verificações de integridade — um cenário comum em software comercial e que também se beneficia de observabilidade em nível de instrução.

Ainda assim, o trabalho revelou uma limitação estrutural que também aparece nos estudos relacionados: a evasão praticada por protetores comerciais é *redundante* por construção. Isso significa que mitigar apenas uma ou duas sondagens não é suficiente; é necessário tratar o conjunto completo de verificações (tempo, ambiente, código mapeado, exceções, depuração, VM) para que o executável não degrade o comportamento observado. Dessa forma, as contramedidas precisam ser igualmente compostas, capazes de atuar em mais de uma camada (instrumentador, sistema operacional, ambiente virtualizado).

Este trabalho reforça a necessidade de investigações contínuas e de novas ferramentas de instrumentação e análise, capazes de lidar com as múltiplas camadas de ofuscação, verificações de *debuggers*, manipulação de exceções e *packers* avançados. A evolução persistente dos *malwares* e dos protetores comerciais demanda uma postura ativa por parte da comunidade de segurança, bem como o desenvolvimento de abordagens integradas — combinando análise estática, dinâmica e heurística — para detectar e mitigar esses comportamentos complexos. Os resultados aqui obtidos podem servir de base para aprimorar técnicas já existentes e fomentar pesquisas futuras voltadas ao enfrentamento das ameaças modernas de *malware*.

Um aspecto adicional a ser considerado refere-se ao caráter dual das ferramentas de análise de *malware*. Embora a **Contradef** tenha sido concebida com finalidade defensiva, acadêmica e forense, seus recursos de instrumentação, manipulação de fluxo e mitigação de técnicas evasivas podem, em princípio, ser empregados de forma indevida por agentes mal-intencionados, por exemplo, para estudar mecanismos de proteção e aperfeiçoar novas formas de evasão. Esse dilema ético é inerente a pesquisas em segurança ofensiva e análise de código malicioso: os mesmos conhecimentos que permitem compreender, detectar e mitigar ameaças também podem ser explorados para tornar ataques mais sofisticados.

Por essa razão, o uso da **Contradef** deve ser orientado por princípios de responsabilidade, rastreabilidade e contenção. Sua aplicação deve ocorrer em ambientes controlados e isolados, com amostras devidamente autorizadas para pesquisa, respeitando políticas institucionais, normas legais e boas práticas de divulgação responsável. A disponibilização pública da ferramenta destina-se exclusivamente a fins defensivos, acadêmicos e forenses, sendo expressamente incompatível com usos voltados à criação, aprimoramento, ocultação ou disseminação de código malicioso. Dessa forma, embora não seja possível controlar integralmente o uso posterior de uma ferramenta pública, este trabalho delimita seu propósito legítimo e condena qualquer aplicação mal-intencionada de seus mecanismos.

Esta dissertação mostrou que é viável usar DBI para revelar e registrar comportamento evasivo em *malware* para Windows, inclusive quando protegido por empacotadores comerciais, *desde que* o próprio ambiente de instrumentação seja tratado como alvo e receba camadas de proteção. Os artefatos e módulos produzidos podem servir como base para evoluções futuras da

ferramenta e para estudos comparativos que busquem, com dados empíricos, reduzir o atual hiato entre o que a literatura propõe e o que, de fato, é executável contra amostras evasivas no mundo real. Ao mesmo tempo, reconhece-se que tais avanços devem ser conduzidos com responsabilidade ética, de modo que o conhecimento produzido seja direcionado prioritariamente ao fortalecimento de capacidades defensivas, à melhoria de processos forenses e ao desenvolvimento de mecanismos mais eficazes de detecção e mitigação de ameaças.

8.2 Contribuições Alcançadas

Este trabalho produziu contribuições de natureza conceitual, técnica e prática para a análise de executáveis Windows protegidos por empacotadores comerciais e dotados de comportamento evasivo. Em particular, destacam-se:

- **Proposição de métodos baseados em DBI** para *detecção* de técnicas de evasão em *malware* e em binários protegidos, com foco em sondagens de ambiente, consultas a APIs sensíveis e padrões de controle de fluxo;
- **Desenvolvimento de rotinas anti-evasão** capazes de normalizar respostas do ambiente e, assim, permitir que o código malicioso avance para o trecho de interesse analítico;
- **Métodos de identificação de protetores comerciais**, por meio de regras *YARA* específicas, capazes de sinalizar a presença de *Themida* e *Obsidium* antes do início da instrumentação;
- **Mecanismo dinâmico de detecção de sequências de instruções em assembly** (*Detector de Sequências de Instruções*) para identificar e, quando necessário, manipular o fluxo de execução em tempo de execução;
- **Disponibilização da ferramenta *ContradeF*** (código e estratégias de instrumentação) para uso e extensão pela comunidade de segurança, favorecendo reprodutibilidade e comparações futuras;
- **Artigo submetido e aceito** no Salão de Ferramentas do XV Simpósio Brasileiro de Cibersegurança (SBSeg), apresentando a *ContradeF* e premiado como **melhor ferramenta** do evento, o que reforça a relevância e a viabilidade da proposta.

8.3 Trabalhos Futuros

Os resultados apontam para cinco frentes principais de continuidade:

- **Integração de análises:** combinar instrumentação dinâmica com análise estática e com detectores por assinatura/heurística, de modo a antecipar pontos de evasão e reduzir falsos positivos.
- **Expansão e atualização das contramedidas:** incorporar contramedidas para técnicas mais recentes de anti-DBI e acompanhar novas versões de protetores comerciais (*Themida*, *Obsidium*, *VMProtect*) e de alvos x86 (32 bits).
- **Instrumentação adaptativa e *logging* enxuto:** adotar um perfil de instrumentação que se ajuste dinamicamente ao comportamento observado e incluir estratégias para reduzir o volume de *logs* e *traces*.

- **Adoção de um formato estruturado de saída:** produzir saídas estruturadas (p.ex. JSON) que facilitem a correlação e comparação entre execuções da **Contradef**, viabilizando também o processamento por ferramentas automatizadas.
- **Identificação de protetores por aprendizagem de máquina:** investigar o uso de técnicas de aprendizagem de máquina para reconhecer padrões ou assinaturas comportamentais de protetores comerciais a partir dos artefatos coletados pela **Contradef**, como sequências de APIs, padrões de instruções, acessos à memória, eventos temporais e estruturas do binário. Essa abordagem pode complementar as regras *YARA*, permitindo identificar variantes ou versões modificadas de protetores mesmo quando assinaturas estáticas tradicionais não forem suficientes.

Referências Bibliográficas

- Afianian, A., Niksefat, S., Sadeghiyan, B., and Baptiste, D. (2018). Malware dynamic analysis evasion techniques: A survey. *arXiv arXiv:1811.01190*.
- Arafa, P., Kashif, H., and Fischmeister, S. (2013). Dime: Time-aware dynamic binary instrumentation using rate-based resource allocation. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE.
- Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., and Vigna, G. (2010). Efficient detection of split personalities in malware. In *NDSS*.
- Basili, V. R., Caldiera, G., and Rombach, H. D. (2010). *The Goal Question Metric Approach*. Wiley.
- Behera, C. K. and Bhaskari, D. L. (2015). Different obfuscation techniques for code protection. *Procedia Computer Science*, 70:757–763.
- Bernat, A. R., Roundy, K., and Miller, B. P. (2011). Efficient, sensitivity resistant binary instrumentation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 89–99.
- Botacin, M., da Rocha, V. F., de Geus, P. L., and Grégio, A. (2017). Analysis, anti-analysis, anti-anti-analysis: an overview of the evasive malware scenario. *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*, pages 250–263.
- Branco, R. R., Barbosa, G. N., and Drimel, P. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. "<https://api.semanticscholar.org/CorpusID:53680184>". Acessado em: 28-08-2024.
- Bruening, D., Zhao, Q., and Amarasinghe, S. (2012). Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 133–144.
- Cao, Y., Liang, R., Chen, K., and Hu, P. (2022). Boosting neural networks to decompile optimized binaries. In *proceedings of the 38th annual computer security applications conference*, pages 508–518. Association for Computing Machinery.
- Check Point Research (2022a). Anti-debug: Debug flags. "<https://anti-debug.checkpoint.com/techniques/debug-flags.html>". Acessado em: 14-01-2025.
- Check Point Research (2022b). Anti-debugging: Misc. techniques. "<https://anti-debug.checkpoint.com/techniques/misc.html>". Acessado em: 09-01-2025.

- Chen, X., Andersen, J., Mao, Z. M., Bailey, M., and Nazario, J. (2008). Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 177–186. IEEE.
- Cheng, B., Ming, J., Leal, E. A., Zhang, H., Fu, J., Peng, G., and Marion, J.-Y. (2021). Obfuscation-Resilient executable payload extraction from packed malware. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3451–3468. USENIX Association.
- Coccia, G., Polino, M., Carminati, M., and Zanero, S. (2022). A study of evasive behaviors in commercial packers. Dissertação (mestrado), Politecnico di Milano, Milan, Italy.
- Crăciun, V., Mogage, A., and Lucanu, D. (2023). Full transparency in DBI frameworks. *arXiv preprint arXiv:2306.13529*.
- D'Alessio, S. and Mariani, S. (2015). PinDemonium: a DBI-based generic unpacker for Windows executables. Dissertação (mestrado), Politecnico di Milano, Milan, Italy.
- DeceptiveBytes (2018). Tricks used by malware authors to protect their malicious code from detection. "<https://deceptivebytes.com/tricks-used-by-malware-authors-to-protect-their-malicious-code-from-detection/>". Acessado em: 24-10-2025.
- D'Elia, D. C., Coppa, E., Nicchi, S., Palmaro, F., and Cavallaro, L. (2019). SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 15–27.
- Distributed Management Task Force (DMTF) (2025). SMBIOS | DMTF. "<https://www.dmtf.org/standards/smbios>". Acessado em: 20-10-2025.
- Duncan Ogilvie (2022). TitanHide: Hiding kernel-driver for x86/x64. "<https://github.com/mrexodia/TitanHide>". Acessado em: 10-01-2025.
- DynamoRIO (2024). DynamoRIO - dynamic instrumentation tool platform - overview. "https://dynamorio.org/page_home.html". Acessado em: 21-07-2024.
- D'Elia, D. C., Coppa, E., Palmaro, F., and Cavallaro, L. (2020). On the dissection of evasive malware. *IEEE Transactions on Information Forensics and Security*, 15:2750–2765.
- D'Elia, D. C., Invidia, L., Palmaro, F., and Querzoni, L. (2022). Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts. *Digital Threats: Research and Practice (DTRAP)*, 3(2):1–13.
- Falcón, F. and Riva, N. (2012). Dynamic binary instrumentation frameworks: I know you're there spying on me. In *Proceedings of the RECon 2012*. CORE Security.
- Ferrie, P. (2011). The ultimate anti-debugging reference. "<https://pferrie.epizy.com/papers/antidebug.pdf>". Acessado em: 23-09-2024.
- Filho, A. S. (2019). Reduzindo a superfície de ataque dos frameworks de instrumentação binária dinâmica. Dissertação (mestrado), Universidade Federal do Amazonas, Manaus, AM.
- Filho, A. S., Rodríguez, R. J., and Feitosa, E. L. (2022). Evasion and countermeasures techniques to detect dynamic binary instrumentation frameworks. *Digital Threats: Research and Practice*, 3.

- FreeFixer / SystemExplorer (2024). What is cmdvrt32.dll? "<https://www.freefixer.com/library/file/cmdvrt32.dll-197227/>". Acessado em: 20-10-2025.
- Frida Docs (2024). Frida docs - getting started. "<https://frida.re/docs/home/>". Acessado em: 20-07-2024.
- Galloro, N., Polino, M., Carminati, M., Continella, A., and Zanero, S. (2022). A systematical and longitudinal study of evasive behaviors in windows malware. *Computers & security*, 113:102550.
- Gozzini, S., Polino, M., Carminati, M., and Zanero, S. (2022). PINvader: a dynamic analysis tool for evasive techniques detection and bypass in 64-bit Windows binaries. Dissertação (mestrado), Politecnico di Milano, Milan, Italy.
- Hendrikse, S. and Austen, J. (2012). Malware armouring: The case against incident-related binary analysis. Technical report, Royal Holloway. "https://cdn.ttgtmedia.com/rms/pdf/RH_IncidentResponse.pdf". Acessado em: 17-07-2024.
- Hoglund, G. and McGraw, G. (2003). Reverse engineering and program understanding. Technical report, InformIT. Excerpt from the book: Exploiting Software: How to Break Code. "https://ptgmedia.pearsoncmg.com/imprint_downloads/informit/chap03_0201786958.pdf". Acessado em: 14-07-2024.
- Intel (2024). Pin 3.31 user guide. "<https://software.intel.com/sites/landingpage/pintool/docs/98861/Pin/doc/html/index.html>". Acessado em: 21-07-2024.
- Joseph Landry (2016). Malware discovered – SFG: Furtim malware analysis. "<https://www.sentinelone.com/blog/sfg-furtims-parent/>". Acessado em: 24-10-2025.
- Kim, J.-W., Bang, J., and Choi, M.-J. (2020). Defeating anti-debugging techniques for malware analysis using a debugger. *Advances in Science, Technology and Engineering Systems Journal*, 5(6):1178–1189.
- Kim, M., Cho, H., and Yi, J. (2022). Large-scale analysis on anti-analysis techniques in real-world malware. *IEEE Access*, 10:1–1.
- Kirat, D., Vigna, G., and Kruegel, C. (2011). BareBox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC '11*, page 403–412, New York, NY, USA. Association for Computing Machinery.
- Kitchenham, B. A. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical report, EBSE, Keele University. "https://legacyfileshare.elsevier.com/promis_misc/525444systematicreviewsguide.pdf". Acessado em: 14-07-2024.
- Kruegel, C. (2014). Full system emulation: Achieving successful automated dynamic analysis of evasive malware. In *Proceedings of the BlackHat USA Security Conference*, pages 1–7. Presented at BlackHat USA Security Conference.
- Lee, G., Kim, M., Yi, J. H., and Cho, H. (2024). Pinicorn: Towards automated dynamic analysis for unpacking 32-bit PE malware. *Electronics*, 13(11):2081.
- Lee, Y. B., Suk, J. H., and Lee, D. H. (2021). Bypassing anti-analysis of commercial protector methods using DBI tools. *IEEE Access*, 9:7655–7673.

- Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200.
- Martins, G. B. (2016). *Identificação de Malware Metamórfico baseado em Grafos de Dependência*. Tese de doutorado, Universidade Federal do Amazonas, Manaus, Brasil. "<http://tede.ufam.edu.br/handle/tede/5641>". Acessado em: 05-05-2026.
- Microsoft Learn (2023a). Função CreateFileW. <https://learn.microsoft.com/pt-br/windows/win32/api/fileapi/nf-fileapi-createfilew>. Acessado em: 11-01-2025.
- Microsoft Learn (2023b). Função NtQueryInformationProcess. "<https://learn.microsoft.com/pt-br/windows/win32/api/winternl/nf-winternl-ntqueryinformationprocess>". Acessado em: 14-01-2025.
- Microsoft Learn (2024a). Checkremotedebuggerpresent function (debugapi.h). "<https://learn.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-checkremotedebuggerpresent>". Acessado em: 14-01-2025.
- Microsoft Learn (2024b). FindWindowA function (winuser.h). "<https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-findwindowa>". Acessado em: 22-10-2025.
- Microsoft Learn (2024c). Função GetModuleHandleExW (loaderapi.h). "<https://learn.microsoft.com/pt-br/windows/win32/api/loaderapi/nf-loaderapi-getmodulehandleexw>". Acessado em: 20-10-2025.
- Microsoft Learn (2024d). Nomear arquivos, caminhos e namespaces. "<https://learn.microsoft.com/pt-br/windows/win32/fileio/naming-a-file>". Acessado em: 11-01-2025.
- Microsoft Learn (2025). Função NtQuerySystemInformation (winternl.h). "<https://learn.microsoft.com/pt-br/windows/win32/api/winternl/nf-winternl-ntquerysysteminformation>". Acessado em: 24-10-2025.
- Microsoft Sysinternals (2025). Process monitor (Procmon). "<https://learn.microsoft.com/en-us/sysinternals/downloads/procmon>". Acessado em: 22-10-2025.
- Mohanta, A. and Saldanha, A. (2020). *Malware analysis and detection engineering: a comprehensive approach to detect and analyze modern malware*. Springer.
- Muralidharan, T., Cohen, A., Gerson, N., and Nissim, N. (2022). File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements. *ACM Computing Surveys*, 55(5):1–45.
- Olaimat, M. N., Maarof, M. A., and Al-rimy, B. A. S. (2021). Ransomware anti-analysis and evasion techniques: A survey and research directions. In *2021 3rd international cyber resilience conference (CRC)*, pages 1–6. IEEE.
- Or-Meir, O., Nissim, N., Elovici, Y., and Rokach, L. (2019). Dynamic malware analysis in the modern era—a state of the art survey. *ACM Computing Surveys (CSUR)*, 52(5):1–48.
- Oreans (2025). Virtual machine. "https://www.oreans.com/help/tm/hm_virtual-machine.htm". Acessado em: 10-10-2025.

- Paradyn (2008). Paradyn parallel performance tools - dyninst programmer's guide. "<https://pages.cs.wisc.edu/~roundy/coverage/dyninstGuide.pdf>".
- Park, J., Jang, Y. H., Hong, S., and Park, Y. (2019). Automatic detection and bypassing of anti-debugging techniques for microsoft Windows environments. *Advances in Electrical and Computer Engineering*, 19:23–28.
- Park, S. and Park, Y. (2024). Detection techniques for DBI environment in Windows. *Electronics (Switzerland)*, 13(5):871.
- Polino, M., Continella, A., Mariani, S., D'Alessio, S., Fontana, L., Gritti, F., and Zanero, S. (2017a). Measuring and defeating anti-instrumentation-equipped malware. In Polychronakis, M. and Meier, M., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 73–96, Cham. Springer International Publishing.
- Polino, M., Continella, A., Mariani, S., D'Alessio, S., Fontana, L., Gritti, F., and Zanero, S. (2017b). Hiding pin's artifacts to defeat evasive malware. In *Black Hat Europe 2017*.
- Quarkslab (2024). QDBI - quarkslab dynamic binary instrumentation. "<https://qbdi.quarkslab.com/>". Acessado em: 20-07-2024.
- Rodríguez, R. J., Artal, J. A., and Merseguer, J. (2014). Performance evaluation of dynamic binary instrumentation frameworks. *IEEE Latin America Transactions*, 12(8):1572–1580.
- Rodríguez, R. J., Gaston, I. R., and Alonso, J. (2016). Towards the detection of isolation-aware malware. *IEEE Latin America Transactions*, 14(2):1024–1036.
- Sandboxie-Plus Project (2024a). Code injection — Sandboxie documentation. "<https://sandboxie-plus.github.io/sandboxie-docs/Content/CodeInjection/>". Acessado em: 20-10-2025.
- Sandboxie-Plus Project (2024b). Isolation mechanism — Sandboxie documentation. "<https://sandboxie-plus.github.io/sandboxie-docs/Content/IsolationMechanism/>". Acessado em: 20-10-2025.
- Sandboxie-Plus Project (2024c). SBIE DLL API. "<https://sandboxie-plus.com/sandboxie/sbiedllapi/>". Acessado em: 20-10-2025.
- Santos Filho, A., Alves, A. B., Vieira, I., and Feitosa, E. L. (2017). Mitigando a evasão de instrumentadores binários dinâmicos. In *Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg)*, pages 479–486. SBC.
- Shvets, A. (2023). Padrão de projeto Strategy. "<https://refactoring.guru/pt-br/design-patterns/strategy>". Acessado em: 15-10-2024.
- Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA.
- Singh, J. and Singh, J. (2018). Challenge of malware analysis: Malware obfuscation techniques. *International Journal of Information Security Science*, 7(3):100–110.
- Skaletsky, A., Devor, T., Chachmon, N., Cohn, R., Hazelwood, K., Vladimirov, V., and Bach, M. (2010). Dynamic program analysis of Microsoft Windows applications. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 2–12. IEEE.

- Soriano-Salvador, E. and Guardiola-Múzquiz, G. (2023). Detecting and bypassing Frida dynamic function call tracing: exploitation and mitigation. *Journal of Computer Virology and Hacking Techniques*, 19(4):503–513.
- Suk, J. H., Lee, J.-Y., Jin, H., Kim, I. S., and Lee, D. H. (2018). UnThemida: Commercial obfuscation technique analysis with a fully obfuscated program. *Software: Practice and Experience*, 48(12):2331–2349.
- UEFI Forum, Inc. (2021). Advanced configuration and power interface (ACPI) specification. "https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/". Acessado em: 22-10-2025.
- Vishnani, K., Pais, A. R., and Mohandas, R. (2011). Detecting & defeating split personality malware. In *The Fifth International Conference on Emerging Security Information, Systems and Technologies*, pages 7–13. IEEE Computer Security Los Alamitos.
- Xu, F., Zhang, W., Qiang, W., and Jin, H. (2023). Multi-path exploration guided by taint and probability against evasive malware. *Security and Safety*, 2:2023023.
- Yason, M. V. (2007). The art of unpacking. "<https://blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf>". Acessado em: 10-09-2024.
- Zhang, B. (2021). Research summary of anti-debugging technology. In *Journal of Physics: Conference Series*, volume 1744. IOP Publishing Ltd.
- Zhang, S., Dong, W., and Lin, J. (2025). Devmp: A virtual instruction extraction method for commercial code virtualization obfuscators. In *Proceedings of the 16th International Conference on Internetware*, Internetware 25, pages 258–268, New York, NY, USA. Association for Computing Machinery.