
# Contradef: Uma Ferramenta de Instrumentação Binária Dinâmica para Análise de Malware Evasivo

Henrique B. Campelo[1], Francisco S. S. Neto[1], Eduardo L. Feitosa[1],

[1]Instituto de Computação – (ICOMP) – Universidade Federal do Amazonas (UFAM)
CEP – 69.077-000 – Manaus – AM – Brasil

{henrique.borges,francisconeto,efeitosa}@icomp.ufam.edu.br

***Abstract.*** *Contradef is a DBI tool built on Intel Pin. It is designed for analyzing evasive software through tracing techniques. The tool records instruction flows, memory accesses, API calls, and internal states to log files, enabling post-execution analysis. Contradef enables execution, inspection, and even manipulation of software protected by advanced packers like VMProtect, revealing obfuscation and evasion techniques that typically bypass static analysis methods.*

***Resumo.*** *A Contradef é uma ferramenta DBI, desenvolvida sobre o Intel Pin, para a análise de software evasivo, por meio de técnicas de tracing. Ela registra, em arquivos, o fluxo de instruções, os acessos à memória, as chamadas de API e outros estados internos, permitindo que esses dados sejam investigados depois da execução. Desta forma, a análise desses registros permite revelar técnicas de ofuscação e evasão, empregadas por empacotadores de software, como o VMProtect.*

## 1. Introdução

A evolução dos *malwares* tem sido acompanhada pelo uso crescente de técnicas de anti-análise, como empacotamento, criptografia e evasão baseada no ambiente de execução, com o intuito de dificultar a inspeção por ferramentas automatizadas [Coccia et al. 2022, Zhang et al. 2023]. Nesse cenário, a Instrumentação Binária Dinâmica (*Dynamic Binary Instrumentation* – DBI) destaca-se como uma alternativa relevante, ao permitir a análise de binários em tempo real, sem requerer acesso ao código-fonte nem recompilação. Ao possibilitar a inspeção de registradores, regiões de memória e chamadas ao sistema operacional durante a execução, DBI oferece meios para observar de forma precisa o comportamento interno de binários potencialmente maliciosos. Ferramentas como o Intel Pin [Intel 2024] viabilizam a instrumentação em nível de instrução, permitindo o monitoramento detalhado de chamadas ao sistema, acessos à memória e padrões de execução.

No entanto, têm surgido mecanismos capazes de detectar a presença de instrumentação ativa, o que torna as ameaças ainda mais sofisticadas. Essas técnicas exploram artefatos introduzidos pelos instrumentadores na memória do processo, padrões de execução atípicos e estruturas de ambiente não usuais. Uma vez detectada a instrumentação, o código malicioso pode interromper sua atividade, alterar seu fluxo de controle ou adotar comportamentos artificiais, comprometendo a eficácia da análise dinâmica [Rodríguez et al. 2016, Polino et al. 2017].

Este artigo apresenta a **Contradef**[1], uma ferramenta desenvolvida sobre o framework Intel Pin, voltada à análise de binários para sistemas operacionais Windows 10 ou superiores, na arquitetura x64, que incorporam mecanismos de evasão. Ela realiza rastreamento contínuo e configurável de instruções e eventos do sistema, com base em *callbacks* especializados, coletando evidências e indícios de comportamentos evasivos. Em experimentos com amostras protegidas por empacotadores comerciais, como o VMProtect, ela foi capaz de identificar características evasivas que frequentemente não são detectadas por abordagens tradicionais. O Intel Pin foi adotado por ser o framework DBI mais utilizado no Windows, principal alvo de malwares.

## 2. Arquitetura da ContraDef

A **Contradef** foi desenvolvida com foco na análise de binários protegidos por técnicas de evasão. Sua arquitetura, disponível no repositório da ferramenta, é composta por cinco (5) módulos especializados, projetados para funcionar tanto de forma autônoma como atuar de maneira coordenada durante processos de engenharia reversa aplicados a binários protegidos. Cada módulo é responsável por lidar com um conjunto específico de técnicas de análise, oferecendo mecanismos para inspeção de regiões de memória, rastreamento de instruções e interceptação de chamadas ao sistema operacional. A saída de cada módulo, em texto puro, contêm campos específicos do modo ativado (como parâmetros e retornos de APIs, endereços e valores de memória), permitindo a correlação entre módulos e a inspeção posterior.

### 2.1. FunctionInterceptor

O módulo *FunctionInterceptor* implementa um mecanismo de *hooking* seletivo para interceptar invocações a funções sensíveis do sistema operacional, como, por exemplo, $GetProcAddress$, $VirtualProtect$ e $NtQueryInformationProcess$. Para cada chamada interceptada, são registrados (Figura 1) o identificador da *thread* (*Thread 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.

```
CheckRemoteDebuggerPresent...
   Nome do modulo: C:\WINDOWS\System32\KERNELBASE.dll
   Thread: 0
   Id de chamada: 0
   Endereco da rotina: 7ffde01f93a0
   Parametros: (hProcess: 0; pbDebuggerPresent: FALSE)
   Valor de retorno: 1
```

**Figura 1. Captura de informações da função *CheckRemoteDebuggerPresent*.**

### 2.2. Trace de Funções Externas (*TraceFcnCall*)

O módulo *TraceFcnCall* registra a sequência cronológica de cada função invocada, identificando endereço, `DLL`, nome simbólico da função e a *thread* responsável pela chamada (Figura 2). O resultado fornece uma visão geral do fluxo de execução, útil para segmentar a amostra em fases distintas, como instalação, persistência, comunicação em rede ou sabotagem. Padrões comportamentais, como, por exemplo, sequências repetidas de chamadas a `Sleep`, indicativas de atraso de execução ou ociosidade deliberada, tornam-se evidentes no log e permitem isolar rotinas suspeitas para investigação aprofundada.

[1] `https://github.com/contradef/Contradef`

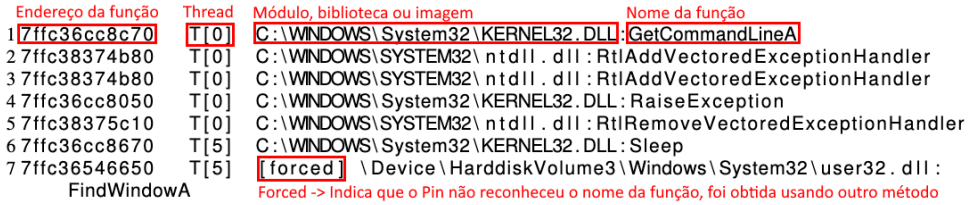


**Figura 2. Exemplo da lista de funções interceptadas.**

### 2.3. TraceMemory

O módulo *TraceMemory* monitora operações de leitura e escrita em memória, registrando o endereço acessado, a instrução responsável pela operação, o valor original e o novo valor armazenado (limitado a 16 bytes) - (Figura 3). Ele também destaca alterações de permissão de páginas de memória que transitam do estado `RW` (leitura/escrita) para `RX` (leitura/execução), um indício recorrente de auto-desempacotamento em amostras protegidas por empacotadores. O rastreamento de memória, ao registrar o endereço, a instrução e o conteúdo dos dados acessados em tempo de execução, permite a extração de *strings* decifradas contendo nomes de arquivos, janelas, APIs, domínios de C2 e artefatos indicativos de mecanismos de anti-análise.

```
------------------------------------------------------------------------
[T0] 0x00007ff635935297  |  movzx rsi, word ptr [rsi] | [Read 1Op]:
        [0x00007ff6357932d5] = 0x0144 (324)
------------------------------------------------------------------------
[T0] 0x00007ff6359352db  |  mov rsi, qword ptr [rsi]  | [Read 1Op]:
        [0x00007ff6357933ec] = 0x00007ff635a24908 (140695438510344)
------------------------------------------------------------------------
[T0] 0x00007ff6359352f2  |  mov sil, byte ptr [rsi]   | [Read 1Op]:
        [0x00007ff635a24908] = 0x50 (80) | "PROCMON_WINDOW_CLASS"
------------------------------------------------------------------------
```

**Figura 3. Extrato Capturado pelo TraceMemory.**

### 2.4. TraceInstructions

O módulo *TraceInstructions* registra, de forma sequencial, cada endereço e instrução executada, juntamente com os valores dos registradores, os *flags* e os operandos lidos ou modificados (Figura 4).

```
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             |
```

**Figura 4. Extrato Capturado pelo TraceInstructions.**

### 2.5. TraceDisassembly

O módulo *TraceDisassembly* percorre cada bloco de código, à medida que o Pin o descobre, em tempo de execução, e gera um relatório contínuo das instruções já traduzidas para *assembly* simbólico, com endereços e operandos resolvidos (Figura 5). Vale destacar que o módulo apenas registra o fluxo que o binário pretende seguir, sem que a instrução tenha sido efetivamente executada.

O resultado é um "mapa" dos caminhos percorridos, útil para localizar rotinas já desempacotadas ou identificar saltos e chamadas de API. Quando o analista precisa de dados dinâmicos, como valores em registradores ou conteúdo de memória, esse código desmontado pode ser combinado a módulos mais granulares como o *TraceInstructions* ou o *TraceMemory*, mas ele permanece a opção ideal para obter rapidamente a topologia do binário sem o volume de um rastreio completo de instruções.

```
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
```

**Figura 5. Extrato Capturado pelo TraceDisassembly.**

## 3. Limitações

Apesar dos benefícios apresentados, a *Contradef* possui algumas restrições práticas: (1) suporte restrito a executáveis PE 64-bit; (2) dependência de operações de entrada/saída em disco; (3) geração de tamanhos excessivos dos arquivos de rastreamento; (4) sensibilidade a técnicas de evasão anti-DBI capazes de detectar aumento de *overhead*; (5) necessidade de interrupção manual em execuções prolongadas; (6) variabilidade no tempo de execução, de acordo com o perfil de instruções do binário analisado. Vale destacar que elas não comprometem a utilidade da *Contradef*, mas exigem precauções operacionais adicionais, como a escolha criteriosa dos modos de instrumentação, a provisão de armazenamento compatível com o volume de dados esperado e, sempre que possível, a execução em ambientes com alto desempenho.

## 4. Experimentos

### 4.1. Ambiente de Experimentação

Todos os ensaios foram conduzidos em ambiente virtualizado controlado, com o objetivo de isolar riscos e garantir a reprodutibilidade dos resultados. A máquina hospedeira dispõe de um processador Intel Core i7-1165G7, 16 GB de RAM e uma unidade SSD SATA III Crucial CT1000BX500SSD1 de 1 TB. O *hypervisor* adotado foi o Oracle VirtualBox 7.0.

**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.

**Conjunto de amostras.** Foram utilizadas amostras reais de *malware* x64 conhecidas por empregar técnicas de evasão. As amostras utilizadas foram obtidas no serviço *MalwareBazaar* (`https://bazaar.abuse.ch/`), filtrando executáveis do tipo PE para arquitetura x64. Cada amostra foi então verificada no *VirusTotal* (`https://www.virustotal.com/`) para confirmar a presença de artefatos relacionados a técnicas de evasão. Posteriormente, a presença desses mecanismos foi confirmada por meio da ferramenta *Detect It Easy* (DIE) e por regras YARA específicas, garantindo a presença de empacotadores e artefatos *anti-debug/anti-VM*. As amostras foram mantidas compactadas e armazenadas localmente no repositório da ferramenta. Por questão de espaço, descrevemos apenas duas amostras identificadas como potencialmente maliciosas que foram analisadas pela Contradef.

**Condições de execução.** Para evitar interferências, recursos como o *Windows Defender* e o *User Account Control* foram desabilitados. O terminal do Windows foi iniciado com privilégios administrativos, assegurando que os binários tenham acesso irrestrito às chamadas de sistema. A *Contradef* foi invocada diretamente via `pin.exe`, sendo avaliados: (i) cada modo de instrumentação individualmente; (ii) a execução combinada de

todos os módulos; e (iii) a execução nativa (sem *Contradef*), utilizada como linha de base comparativa.

**Métrica de desempenho.** O tempo de execução total de cada cenário foi medido utilizando o `Measure-Command` do PowerShell. Adicionalmente, o tamanho final dos arquivos de *log* gerados em cada configuração foi registrado, permitindo estimar o impacto em termos de armazenamento.

### 4.2. Resultados

Esta seção apresenta os resultados obtidos com a aplicação da ferramenta Contradef sobre amostras evasivas. Os dados coletados pelos módulos revelam padrões de comportamento malicioso relevantes para a análise dinâmica, como chamadas a APIs sensíveis, manipulação de memória e técnicas de evasão por atraso.

#### 4.2.1. Amostra 1 – 36685efcf34c7a7a6f6dd2e48199e4700b5ab8fe3945a50297703dd8daced74f

Conforme dados consultados no *VirusTotal*, a amostra[2] é classificada como um *Trojan* 64-bit com funcionalidades de persistência, evasão de análise e possível *backdoor*, pertencente às famílias *Ulise*, *Casdet* e *BackdoorX*. Apresenta técnicas anti-debug, verificação de hardware real (CPU e USB), chamadas a WMI, inatividade prolongada (anti-sandbox) e um *overlay* típico de empacotamento.

A ferramenta DIE identificou na amostra o uso do empacotador *VMProtect 2.x* enquanto as regras *YARA* sinalizaram a presença de comportamentos *anti-debug* e alta entropia, indicativo de codificação ou compressão agressiva (regra *IsPacked*). O módulo *FunctionInterceptor* registrou diversas invocações a funções clássicas de detecção de ambientes de análise ($IsDebuggerPresent$, $CheckRemoteDebuggerPresent$ e $NtQueryInformationProcess$) - Figura 6.


**Figura 6. Funções anti-debug identificadas na amostra 1**

Além disso, o parâmetro $ProcessInformationClass$ com valor 30, ativado pela função $NtQueryInformationProcess$, tipicamente é associado à detecção de ambientes *sandbox* ou à evasão de mecanismos de análise, fazendo com que o binário finalize

[2] `https://www.virustotal.com/gui/file/4b29ffb60c0e5f5e9bfecff0061cfa6b`

prematuramente ou entre em um ciclo de inatividade logo após a confirmação da presença do depurador. O mesmo log também evidenciou chamadas consecutivas a $GetTickCount$ e $RtlQueryPerformanceCounter$, indicadores típicos de verificações de *overhead* destinadas a revelar a latência introduzida por ferramentas de análise.

O módulo *TraceInstructions* complementou a análise ao capturar, no endereço `0x00007ff69651cf58`, caracteres que compõem cadeias ASCII reveladas durante o processo de desempacotamento (Figura 7). Esses artefatos sugerem que parte do código foi decifrada em memória durante a execução dos métodos evasivos.

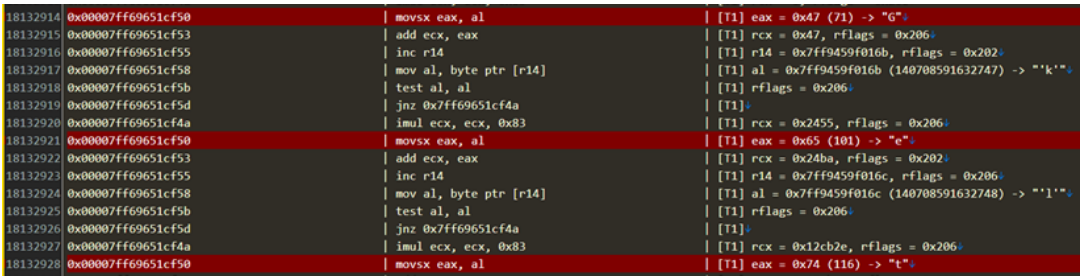


**Figura 7. caracteres que compõem cadeias ASCII desempacotadas**

#### 4.2.2. Amostra 2 – 0f20b0c906f3ad95dbf75ed526b2fe4341fdf62ab8c971fc10e340091af75b3b

A amostra 2 [3] é um Trojan de 64 bits com capacidades de *spyware*, *dropper* e *spreader*. Entre seus comportamentos estão técnicas anti-análise (detecção de VMs e depuradores), inatividade prolongada e uso de *overlay*. Apresenta indícios de persistência no sistema e coleta furtiva de informações, caracterizando uma ameaça projetada para permanência e evasão em ambientes analisados. As regras *YARA* identificaram múltiplos padrões associados a mecanismos de evasão, incluindo *anti_dbg*, *vmdetect* e alta entropia. O binário invocou uma chamada à função `Sleep` após detectar o ambiente de análise, comportamento facilmente identificado no rastreamento de chamadas a funções.

O módulo *FunctionInterceptor* registrou a chamada à função $Sleep$ com o parâmetro $dwMilliseconds = 2000000$ (isto é, 2000 s), configurando um atraso deliberado na execução (Figura 8). Vale destacar que o módulo *TraceFcnCall* revelou que imediatamente após esta suspensão, o programa invoca repetidamente $HeapFree$ e, em seguida, $FatalExit$, indicando liberação de recursos e término prematuro do processo. A sequência sugere que, ao reconhecer um ambiente de análise, o binário ativa um período de atraso deliberado e encerra-se em seguida.

Ainda no módulo *TraceFcnCall*, observamos uma chamada prévia a $EnumSystemFirmwareTables$ , função comumente empregada para detectar hipervisores. Ao buscarmos por esta função no log do *TraceInstructions*, localizamos uma rotina de retorno, seguida de duas comparações contra as cadeias de caracteres `"WAET"` e `"HPET"`. A presença da assinatura `WAET` (*Windows ACPI Emulated Devices*) é reconhecida como evidência de execução dentro de máquinas virtuais [Peterson and Khoury 2023]. O módulo também revelou várias escritas de caracteres que conformam a cadeia literal "*detected*" que correspondem ao texto apresentado no bloco da função $WriteFile$ do log registrado pelo *FunctionInterceptor* (Figura 9).

[3] `https://www.virustotal.com/gui/file/27f5bc346d0a26445a14f135b1c98ecc`

```
[+] SleepEx...
    Thread: 0
    Id de chamada: 0
    Endereço da rotina: 0x7ff9443aa110
    Parâmetros:
        dwMilliseconds: 2000000
        bAlertable: FALSE
    Endereço da função chamante: 0x7ff6195e5c17
  [-] Início da chamada SleepEx
    Retorno SleepEx: 0
    O tempo expirou e a função retornou normalmente.
  [-] Chamada SleepEx concluída
[*] Concluído
```

**Figura 8. Log da invocação à *Sleep* registrado pelo *FunctionInterceptor*.**

```
[+] WriteFile...
    Thread: 0
    Id de chamada: 0
    Endereço da rotina: 0x7ff944393e30
    Tipo de Handle: File (no name available)
    Parâmetros:
        hFile: 0x8c8
        lpBuffer: 0x497854e010
        nNumberOfBytesToWrite: 85 bytes
        lpNumberOfBytesWritten: 0x497854e000
        lpOverlapped: 0x0
    Endereço da função chamante: 0x7ff6195f9ebd
  [-] Início da chamada WriteFile
    lpBuffer (str): acpi sandbox detect by huoji 2023.6.19
[detected] vm-guest detected by table size

    Retorno WriteFile: TRUE
    Bytes escritos: 0 bytes
  [-] Chamada WriteFile concluída
[*] Concluído
```

**Figura 9. Detecção de VM capturada no *FunctionInterceptor*.**

### 4.2.3. Discussão

A avaliação das duas amostras revelou que a **Contradef** apresenta maior valor de análise quando seus módulos de instrumentação são ativados em conjunto. O módulo *FunctionInterceptor*, que realiza interceptação seletiva de APIs, forneceu os primeiros indícios de ações sensíveis. O registro cronológico dessas chamadas, mantido pelo *TraceFcnCall*, situou tais ações no tempo, facilitando inferências sobre o comportamento global do binário. O *TraceMemory* acrescentou evidências sobre dados em tempo de execução e o *TraceInstructions* revelou exatamente que ramo de execução foi seguido, enriquecendo a análise com detalhes sobre valores de registradores e *flags*. Quando essas perspectivas se cruzam, suspeitas em nível macro podem ser validadas, refinadas ou investigadas em profundidade, acelerando a compreensão de algumas das táticas evasivas empregadas.

Na Tabela 1 é apresentado as sobrecargas de tempo e espaço durante a execução das duas amostras. Na amostra 1, a instrumentação completa durou aproximadamente 12 minutos, gerando perto de 4,8 GB; já a amostra 2 durou quase 36 minutos e gerou 883 MB. O aumento temporal da amostra 2 se deve a uma chamada deliberada a $Sleep(2000000)$ (aproximadamente 33 min), típica de atraso de execução ou ociosidade deliberada para frustrar depuradores. Sem esse atraso, o tempo seria comparável ao da amostra 1. Também se observou que o *TraceInstructions* é, de longe, o principal gerador de volume, somando cerca de 5,1 GB (ambas as amostras).

**Tabela 1. Consumo de tempo e espaço de cada módulo nas amostras avaliadas.**

| Modo ativado | Amostra 1 | | Amostra 2 | |
|---|---|---|---|---|
| | **Tempo** | **Tamanho** | **Tempo** | **Tamanho** |
| Sem Contradef | 2 ms | – | 2 ms | – |
| FunctionInterceptor | 15 s 251 ms | 123 KB | 33 min 32 s 987 ms | 78 KB |
| TraceFcnCall | 18 s 959 ms | 59 KB / 98 KB | 33 min 31 s 400 ms | 29 KB / 40 KB |
| TraceMemory | 6 min 2 s 478 ms | 856 MB | 33 min 36 s 565 ms | 14 MB |
| TraceInstructions | 6 min 55 s 118 ms | 4.22 GB | 35 min 49 s 826 ms | 902 MB |
| TraceDisassembly | 14 s 434 ms | 218 KB | 33 min 25 s 420 ms | 329 KB |
| Todos os módulos | 11 min 53 s 527 ms | 4.75 GB | 35 min 42 s 152 ms | 918 MB |

Em síntese, a escolha seletiva de módulos permite balancear duração e volumetria. O tempo de execução da ferramenta varia significativamente de acordo com o comportamento interno do binário analisado e com o volume de dados gerado em cada módulo de instrumentação, especialmente durante a escrita dos arquivos de log. Por esse motivo, os tempos observados não são diretamente comparáveis entre amostras distintas ou aos de outras ferramentas similares.

## 5. Conclusão

A **Contradef** foi projetada para a análise de *malwares* evasivos em ambientes Windows. Embora desenvolvida com foco em programas evasivos, seus recursos são igualmente aplicáveis à inspeção de binários legítimos, sempre que se deseje observar, com alta granularidade, o comportamento interno de um executável. Nossa avaliação experimental confirmou essa premissa: mesmo frente a amostras protegidas por empacotadores como VMProtect, a ferramenta foi capaz de revelar rotinas críticas, extrair artefatos em memória e reconstruir fluxos de controle.

Entre suas principais vantagens destacam-se: (i) a lógica de *hooking* e o Detector de Sequência de Instruções, que permitem redirecionar saltos e alterar `flags` diante de padrões suspeitos; (ii) a utilização de regras *YARA* para restringir a instrumentação a pontos relevantes, minimizando sobrecarga e reduzindo o volume de *logs*; (iii) a ativação condicional dos módulos de rastreamento de memória e instruções, iniciados apenas em eventos de interesse, como a chamada de uma API ou a identificação de uma sequência crítica; e (iv) a capacidade de manipular parâmetros e valores de retorno, forçando a execução de caminhos que o *malware* seguiria apenas sob condições específicas.

Como trabalho futuro, planeja-se adotar um formato estruturado de resultados que facilite a consulta e a correlação de eventos, incluir suporte para executáveis de 32 bits e desenvolver uma abordagem adaptativa, baseada em padrões e heurísticas detectadas dinamicamente, capaz de ajustar automaticamente o perfil de instrumentação conforme o comportamento observado.

## 6. Agradecimentos

Este estudo foi financiado, em parte, pela Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES-PROEX) – Código de Financiamento 001. Este trabalho também contou com o apoio parcial da Fundação de Amparo à Pesquisa do Estado do Amazonas – FAPEAM – por meio do projeto POSGRAD 2024/2025.

## Referências

Coccia, G., Polino, M., Carminati, M., and Zanero, S. (2021-2022). A study of evasive behaviors in commercial packers. Master's thesis, Politecnico di Milano.

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.

Peterson, N. and Khoury, A. (2023). Evading acpi checks in commercial virtualization platforms. `https://revers.engineering/evading-trivial-acpi-checks/`.

Polino, M., Martignoni, L., and Lanzi, A. (2017). Droidtrace: A dynamic analysis tool for android malware detection. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy Workshops (SPW)*, pages 179–185. IEEE.

Rodríguez, P. A., Santos, I., Bringas, P. G., Sanz, B., and Alvarez, G. (2016). Pintracer: A dbi-based framework for tracing malware behavior. In *Proceedings of the 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 107–114. IEEE.

Zhang, Q., Chen, Y., Wang, Y., Wang, S., Fan, Y., and Zeng, Q. (2023). Hermes: A dynamic binary instrumentation framework for the arm platform. *ACM Transactions on Architecture and Code Optimization (TACO)*, 20(1):1–25.