PACKER PARA ARQUIVOS EXECUTÁVEIS DO WINDOWS NA ARQUITETURA x64

PACKER FOR WINDOWS EXECUTABLE FILES ON x64 ARCHITECTURE

Lincoln Roberto Neves de Lima Dias*
Álvaro Ferraz d'Arce**

Resumo

A proteção contra engenharia reversa é crítica no desenvolvimento de software moderno. Os *packers* atuam nesse cenário ofuscando executáveis para dificultar análises estáticas e dinâmicas. Este trabalho apresenta a implementação e validação do "LilithPacker", um packer para a arquitetura Windows x64. A metodologia integrou a análise teórica do formato *Portable Executable* (PE) ao desenvolvimento de um *stub* com técnicas de *anti-debugging* e *anti-disassembly*. Os testes com uma aplicação 3D confirmaram a eficácia do projeto: o software manteve sua performance (*Frames Per Second*) e funcionalidade intactas, reduziu o consumo de memória e atingiu 69,65% de compressão do *payload*. O estudo valida a arquitetura proposta e discute as implicações éticas dessa tecnologia de uso dual.

Palavras-chave: Packer, Executável, x64, Ofuscação, Anti-debug

Abstract

Protection against reverse engineering is critical in modern software development. Packers operate in this scenario by obfuscating executables to hinder static and dynamic analysis. This work presents the implementation and validation of "LilithPacker", a packer for the Windows x64 architecture. The methodology integrated theoretical analysis of the Portable Executable (PE) format with the development of a stub featuring anti-debugging and anti-disassembly techniques. Tests with a 3D application confirmed the project's effectiveness: the software maintained its performance (Frames Per Second) and functionality intact, reduced memory consumption, and achieved 69.65% payload compression. The study validates the proposed architecture and discusses the ethical implications of this dual-use technology.

Keywords*: Packer, Executable, x64, Obfuscation, Anti-debug*

* Aluno do curso de Tecnologia em Análise e Desenvolvimento de Sistemas, da Faculdade de Presidente Prudente. Email: lincoln.dias01@fatec.sp.gov.br

** Álvaro Ferraz d'Arce, Me. em Ciência da Computação, da Faculdade de Presidente Prudente. E-mail: alvaro.ferraz@fatec.sp.gov.br

1. INTRODUÇÃO

O desenvolvimento de software moderno enfrenta um desafio persistente e crítico: a proteção da propriedade intelectual e a garantia da integridade do código. Em um cenário onde a engenharia reversa e a cópia não autorizada podem comprometer modelos de negócio inteiros, o uso de ferramentas de proteção torna-se indispensável. Nesse contexto, os *packers* emergem como mecanismos fundamentais. Estas ferramentas transformam executáveis, comprimindo e criptografando seu conteúdo original (*payload*) e anexando uma rotina de carregamento (*stub*) responsável por restaurar o programa em memória em tempo de execução. Embora amplamente utilizados para proteção legítima de direitos autorais, os *packers* também figuram no centro de discussões de segurança ofensiva e defensiva, sendo frequentemente empregados por malwares para evadir detecções baseadas em assinatura e dificultar a análise por pesquisadores de segurança.

Apesar da abundância de documentação sobre a arquitetura x86 (32 bits), a transição para a arquitetura Windows x64 introduziu complexidades significativas que elevam a barreira de entrada para o desenvolvimento de ferramentas de proteção customizadas. O formato *Portable Executable* (PE) em 64 bits possui particularidades estruturais rigorosas, como cabeçalhos expandidos, tratamento diferenciado de relocações e um *layout* de seções que interage diretamente com mecanismos modernos de segurança do sistema operacional. O problema central reside na escassez de implementações de referência que abordem, de forma didática e funcional, o ciclo completo de criação de um *packer* x64 que seja resistente a técnicas de análise dinâmica moderna, sem comprometer a estabilidade ou o desempenho da aplicação original.

Este trabalho apresenta o desenvolvimento, implementação e validação de um packer para executáveis Windows na arquitetura x64, denominado “LilithPacker”. A proposta consiste em combinar uma análise teórica profunda da arquitetura do formato PE com a aplicação prática de técnicas de ofuscação. O projeto detalha a construção de um pipeline de empacotamento que inclui compressão e criptografia do *payload*, além do desenvolvimento de um *stub* robusto capaz de realizar a resolução dinâmica de importações e aplicar técnicas de *anti-debugging*, *anti-disassembly* e detecção de máquinas virtuais.

Por fim, o trabalho valida a arquitetura proposta através de estudos de caso reais, avaliando o impacto no desempenho, consumo de memória e a eficácia das proteções implementadas contra engenharia reversa.

2. FUNCIONAMENTO E TÉCNICAS DE PROTEÇÃO

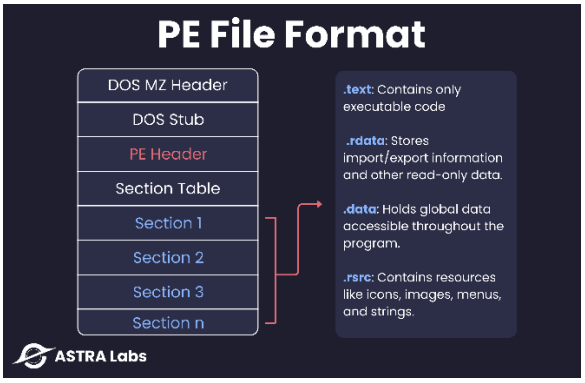
O funcionamento de um *packer* é um processo relativamente complexo, pois envolve diversas etapas que transformam a estrutura original de um executável para protegê-lo contra análise ou modificação. Essa complexidade decorre tanto da necessidade de preservar o correto funcionamento do programa quanto da aplicação de diferentes técnicas de proteção. Entre essas técnicas, podem ser utilizadas abordagens como compressão, criptografia, verificação de integridade e mecanismos anti-análise, cada uma com suas próprias vantagens e limitações. Em conjunto, essas estratégias tornam o *packer* uma ferramenta poderosa para a proteção de software, mas também exigem cuidado e conhecimento técnico para sua implementação eficaz e segura.

2.1 Fundamentação teórica e contexto técnico

O *packer* está intimamente ligado ao formato do binário que se pretende proteger. No ambiente Windows x64, o formato de arquivo utilizado é o PE, que inicia com o cabeçalho DOS (marca “MZ”), seguido da assinatura PE, do *COFF Header* e do *Optional Header*. O *Optional Header* contém campos cruciais para o carregamento, tais como o *ImageBase*, *AddressOfEntryPoint* e o *DataDirectory*, que aponta para tabelas como *Import Table*, *Export Table*, *Relocations*, TLS e *Resource*. A seção de código normalmente reside em “*.text”*, dados somente leitura em “*.rdata”*, e dados mutáveis em .data; arquivos podem conter ainda “*.rsrc”* para recursos e outras seções customizadas.

A Figura 1 ilustra de forma simplificada a composição desse formato, destacando os principais elementos que o compõem e suas respectivas funções.

Figura 1 – Formato de arquivo PE



Fonte: ASTRA Labs (2020). Disponível em: Astra-Labs. Acesso em: 04 nov. 2025.

Em x64, ponteiros de 64 bits e a presença de mecanismos modernos de segurança (ASLR, NX) implicam requisitos específicos quando se modificam endereços e *relocations*[1] durante o empacotamento. Um *packer* é, conceitualmente, um transformador de binários que gera uma nova imagem contendo: um *payload*, o conteúdo original do executável, em geral comprimido e/ou criptografado e um *stub* (também chamado de *loader*), código que, ao ser executado, restaura o *payload* em memória e transfere o controle para o seu ponto de entrada original. Conforme definido por Guo, Ferrie e Chiueh (2008), os *packers* transformam a aparência de um binário de entrada sem afetar sua semântica de execução.

As motivações legítimas para empacotar binários incluem redução de tamanho, proteção de propriedade intelectual e proteção contra adulteração (Assis et al., 2018; Šťastná; Tomášek, 2016); entretanto, as mesmas técnicas são frequentemente empregadas em contextos maliciosos. O uso de *packers* para dificultar a análise estática e evadir a detecção baseada em assinaturas é uma técnica central na ofuscação de *malware* (Guo; Ferrie; Chiueh, 2008; Assis et al., 2018). Embora estudos indiquem que a grande maioria do *malware* é empacotada (Guo; Ferrie; Chiueh, 2008; Assis et al., 2018), a técnica também é marcadamente presente em software legítimo. Por isso, a apresentação técnica de técnicas de proteção deve ser acompanhada de discussão sobre riscos, limitações e uso ético.

[1] *Relocations* são estruturas usadas pelo sistema operacional para ajustar endereços de memória de um executável quando ele não é carregado no endereço base previsto.

2.2 Arquitetura geral do packer proposto

A proposta apresentada neste trabalho organiza o *packer* em três camadas principais:

1. *Parser* e analisador do PE — responsável por ler o binário original, extrair cabeçalhos, identificar seções relevantes, localizar o ponto de entrada e tabelas de importação e *relocations*. Essa etapa produz uma representação interna do *layout* do executável necessário para decidir o que será preservado, removido ou deslocado.
2. *Pipeline* de transformação (empacotamento) — conjunto de operações aplicadas ao *payload*: seleção de seções a empacotar, compressão, criptografia, criação de nova seção para armazenar o *payload* transformado e geração de metadados de carga (tamanho original, offsets, tabela de *relocations* necessária).
3. *Stub* de carregamento (*loader*) — código mínimo que é inserido como novo ponto de entrada do binário final. O *stub* possui a lógica para alocar memória, restaurar o *payload* (descompressão/descrifração), aplicar *relocations* e transferir o controle ao *entrypoint* original. Arquiteturalmente o *stub* deve ser compacto, modular e desenhado para minimizar a superfície de falhas de compatibilidade.

O *pipeline* de empacotamento segue uma sequência lógica e bem definida. Primeiro realiza-se a análise inicial (*parsing*) do executável: lê-se o *Optional Header* e as cabeças de seção do formato PE, identificam-se dependências estáticas através da *Import Table* e localizam-se as entradas de *relocations*. Também verifica-se a presença de uma tabela de certificados; quando o binário está assinado digitalmente, o empacotamento normalmente causa perda da assinatura, exigindo re-assinatura posterior se for necessária a validade do selo digital.

Com a análise concluída, procede-se à seleção do conteúdo a embalar. Nesta etapa decide-se quais seções serão incluídas no *payload* compactado, uma estratégia comum é empacotar *.text* e *.rdata* (código e constantes), ao passo que se mantém *.rsrc* e, em alguns casos, *.data* fora do payload por razões de compatibilidade ou desempenho. A escolha das seções a empacotar influencia diretamente na compatibilidade do binário e no overhead de execução.

Em seguida aplica-se a compressão do *payload*: as seções selecionadas são comprimidas com um algoritmo adequado, para reduzir o tamanho final e alterar a superfície de assinatura estática do arquivo. Esta abordagem, exemplificada por *packers* conhecidos como o UPX, frequentemente envolve a fusão de múltiplas seções em um único bloco de dados comprimido (Guo; Ferrie; Chiueh, 2008). Além de diminuir o espaço ocupado, a compressão tende a aumentar a entropia do *payload*, apresentando uma distribuição de bytes mais uniforme (Šťastná; Tomášek, 2016), o que por si só é um indicador usado em ferramentas de detecção.

Depois da compressão, pode-se aplicar criptografia do *payload*. O bloco comprimido é cifrado com uma cifra simétrica segura (por exemplo AES em modo recomendado ou alternativas modernas como ChaCha20). O gerenciamento da chave é crítico: ela não deve ficar armazenada de forma estática no binário; técnicas de derivação ou proteções em tempo de execução são empregadas para reduzir o risco de extração direta da chave por um analista.

Com *payload* comprimido e cifrado, gera-se um bloco de metadados de carga que descreve as informações necessárias para restaurar o executável em memória: tamanho original descomprimido, *offsets* e mapeamento de seções, mapa de *relocations* e um resumo para verificação de integridade (por exemplo um HMAC). Esses metadados acompanham o *payload* cifrado e fornecem ao *stub* os dados necessários para reconstruir o layout correto do PE em tempo de execução.

O próximo passo é a criação de uma nova seção *.packed* e a alteração do *entrypoint*. No binário final adiciona-se uma seção executável que contém o *stub* de carregamento e os metadados/*payload* empacotados; o campo *AddressOfEntryPoint* do *Optional Header* é atualizado para apontar para o *stub*, de modo que, ao ser executado, o *loader* do Windows inicia o *stub* que fará a restauração do *payload* original em memória.

Por fim realizam-se os ajustes finais e validações: recalculam-se *checksums*[2] quando aplicável, ajustam-se flags e permissões das seções, e verificam-se os alinhamentos exigidos pelo *loader* do Windows para garantir que o binário resultante seja carregável e estável em diferentes ambientes. Testes de compatibilidade e validação de integridade completam o *pipeline* antes de considerar o processo de empacotamento concluído.

[2] *Checksum* é um valor numérico calculado a partir de um conjunto de dados com o objetivo de verificar sua integridade. Se o dado for alterado, o checksum resultante será diferente, permitindo detectar erros ou modificações.

2.3 Design do stub

O *stub* é o componente crítico do *packer*, é o código pelo qual o *Entry Point* da aplicação será trocado, ou seja, o fluxo de execução da aplicação não começará mais pela função "*main*", mas sim pelo código do *stub*: sua função é restaurar o executável original de forma confiável em uma variedade de ambientes, portanto seu desenho exige cuidado em confiabilidade, segurança e compatibilidade.

Uma das primeiras preocupações é a resolução mínima de dependências. O *stub* deve operar com um conjunto restrito de chamadas à API, por exemplo, rotinas para alocação e proteção de memória, e carregamento dinâmico de bibliotecas, e reduzir ao máximo dependências embarcadas para diminuir assinaturas estáticas e aumentar portabilidade entre versões do Windows. Em vez de usar a *Import Table* tradicional, é recomendável que o *stub* resolva dinamicamente endereços de funções essenciais em *runtime*; isso remove *strings* claras do binário e reduz a superfície que ferramentas de análise estática procuram.

A alocação de memória e o gerenciamento de permissões são tarefas centrais do *stub*. Ele deve alocar regiões de memória que inicialmente permitam escrita para copiar e decifrar o *payload* e, em seguida, ajustar as permissões para permitir execução. Em *packers* reais, observa-se a manipulação de permissões de seção em tempo de execução, tanto para a restauração do código original (Guo; Ferrie; Chiueh, 2008) quanto como uma técnica de evasão para confundir analistas (Assis et al., 2018). Boas práticas incluem limpar chaves e buffers sensíveis imediatamente após o uso e minimizar o tempo em que regiões permanecem marcadas como executáveis, medidas que limitam a janela para técnicas de *dump* ou extração por atacantes e analistas.

Após a descompressão/desencriptação, o *stub* precisa realizar a restauração do layout original do PE em memória. Isso implica aplicar corretamente as *relocations* quando o *ImageBase* real em que o processo foi carregado difere do *ImageBase* esperado pelo binário original. Para aplicações que dependem fortemente de *relocations*, a aplicação cuidadosa e ordenada dos blocos de *relocations* é essencial para que ponteiros, tabelas e estruturas internas apontem para os endereços corretos.

A resolução dinâmica de *imports* pode ser implementada de duas formas:

reconstruindo a *Import Table* em memória ou resolvendo funções dinamicamente e preenchendo os ponteiros necessários antes de transferir o controle. Reconstruir a *Import Table* preserva a semântica original e mantém compatibilidade com mecanismos que consultam essas tabelas; já a resolução direta reduz dados estáticos expostos e pode simplificar camadas de ofuscação, embora requeira que o *stub* execute as ligações necessárias de maneira correta e completa.

No momento da transferência de controle e limpeza, depois de restaurado o *payload* e verificadas as checagens de integridade, o *stub* passa a execução para o *AddressOfEntryPoint* original do *payload*. Este processo de transição envolve classicamente a restauração do estado da pilha, por exemplo, garantindo que o ponteiro da pilha (ESP) retorne ao seu valor inicial antes do salto final. Antes dessa transferência é obrigatório que o *stub* elimine das regiões de memória quaisquer estruturas sensíveis (chaves, IVs, buffers temporários) e, quando possível, diminua a superfície de memória executável.

Por fim, o *stub* deve ser projetado para robustez e tolerância a falhas. Deve prever e tratar cenários de exceção, como falha em alocar memória, políticas de segurança do sistema que impeçam operações desejadas, ou incompatibilidades entre versões do OS, implementando rotinas de *fallback* quando viável ou fornecendo mensagens de erro controladas para diagnóstico durante o desenvolvimento. Essas salvaguardas aumentam as chances de o binário permanecer funcional em ambientes diversos sem comprometer a segurança ou a integridade do sistema hospedeiro.

2.4 Técnicas de proteção e seus custos

As técnicas de proteção implementadas em um *packer* têm como objetivo aumentar a resistência do executável frente a engenharia reversa, depuração e análise automatizada, atuando em diferentes camadas, desde a estrutura do arquivo até o comportamento em tempo de execução, sempre buscando equilibrar eficácia, compatibilidade e desempenho.

Entre as principais estratégias, a compressão e criptografia do *payload* visa esconder a imagem original do executável e dificultar a análise estática (Šťastná; Tomášek, 2016). Para isso, as seções críticas do binário, como *.text* e *.rdata*, podem ser comprimidas com algoritmos como LZ4 ou *DEFLATE* e posteriormente cifradas usando cifras simétricas seguras, como

AES em modo autenticado ou ChaCha20. A chave de criptografia geralmente é derivada dinamicamente em tempo de execução a partir de parâmetros variáveis do ambiente, evitando sua exposição em claro no binário. O custo dessa abordagem consiste em aumento de overhead de CPU e latência na inicialização devido à descompressão e desencriptação (Ugarte-Pedrero et al., 2015), além de possíveis dependências ambientais que podem afetar a portabilidade do executável.

As técnicas *anti-debug* buscam detectar a presença de depuradores e dificultar a análise dinâmica (Guo; Ferrie; Chiueh, 2008). Isso é feito por meio de verificações em estruturas do processo, como a checagem da *flag NtGlobalFlag* no *Process Environment Block* (Guo; Ferrie; Chiueh, 2008) ou chamadas diretas a APIs como "*IsDebuggerPresent*". Alguns *packers* também podem implementar essas checagens em threads paralelas ou através de *IAT hooking* para serem executadas periodicamente (Ugarte-Pedrero et al., 2015). Além dessas abordagens, há métodos descritos por Ferrie (2011) que exploram características internas do Windows, como manipulação de *heap flags*, interceptação de exceções e uso de instruções específicas (INT 2D, MOV SS) para detectar o controle de um depurador. Apesar de aumentarem o custo da análise, essas técnicas podem gerar falsos positivos em ambientes legítimos e serem contornadas por depuradores avançados.

Como exemplo demonstrativo, o Quadro 1 apresenta um trecho em Assembly x64 que exemplifica a verificação do campo *NtGlobalFlag* no *Process Environment Block*. O código lê o campo em seu offset apropriado (0xBC), isola os bits comumente definidos quando um processo é criado por um depurador (*FLG_HEAP_ENABLE_TAIL_CHECK*, *FLG_HEAP_ENABLE_FREE_CHECK*, *FLG_HEAP_VALIDATE_PARAMETERS*) e sinaliza a presença de depurador caso a combinação esperada seja encontrada. Essa checagem é compacta, de baixo overhead e não depende de chamadas explícitas a APIs de depuração, o que a torna adequada para execução no *stub* de um *packer*.

Quadro 1 – Trecho em assembly x64 que verifica o campo NtGlobalFlag no PEB para detectar a presença.

```
CheckDebugger_NtGlobalFlag proc
; 1. Obter o endereço do PEB
mov     rax, gs:[60h]
```

```
; 2. Acessar NtGlobalFlag (é um DWORD, 4 bytes)
;    Offset em x64 é 0xBC
mov     eax, [rax + 0BCh]

; 3. Verificar os bits de depuração
and     eax, 0x70

; 4. Se o resultado for 0 (ZF=1), não há depurador.
;    Se for diferente de 0 (ZF=0), um depurador está presente.
jnz     .debugged      ; Jump if Not Zero (debugger detectado)

; Nenhum depurador detectado
xor     rax, rax       ; RAX = 0
ret

.debugged:
; Depurador DETECTADO
mov     rax, 1         ; RAX = 1
ret
CheckDebugger_NtGlobalFlag endp
```

Fonte: *Código em assembly x64 retirado do packer criado pelo autor (2025).*

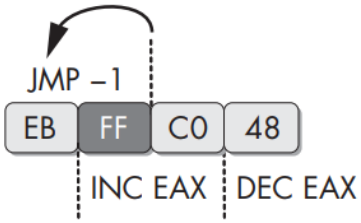
Já as técnicas *anti-disassembly* e *anti-static analysis* visam dificultar a análise estática e a reconstrução do fluxo de controle, utilizando código polimórfico, instruções ambíguas, código auto-modificante e, em níveis avançados, virtualização de código. Nesta última, trechos críticos são traduzidos para um *bytecode* interpretado por uma máquina virtual interna, tornando a análise estática tradicional praticamente impossível (Ugarte-Pedrero et al., 2015). A virtualização de código torna a análise significativamente mais complexa, mas também impacta severamente o desempenho e a compatibilidade do programa (Ugarte-Pedrero et al., 2015).

Entre as técnicas de *anti-disassembly* mais sofisticadas, destaca-se o chamado "*impossible disassembly*", em que bytes individuais são compartilhados por múltiplas instruções que efetivamente são executadas. Esse tipo de código explora limitações dos *disassemblers* tradicionais, tornando impossível representar todas as instruções reais de forma linear ou consistente. O resultado é um fluxo de execução confuso, em que diferentes

caminhos podem se sobrepor no mesmo espaço de bytes, exigindo abordagem manual ou scripts específicos para correção (Sikorski; Honig, 2012).

Um exemplo clássico de implementação da técnica de *impossible disassembly* pode ser observado na Figura 2, onde um simples conjunto de bytes é interpretado de formas diferentes conforme o ponto de entrada do código. Essa ambiguidade proposital faz com que o *disassembler* gere instruções incorretas ou inconsistentes, dificultando a análise estática do executável e mascarando o comportamento real do malware.

Figura 2 – Instrução jump apontando para dentro



Fonte: *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software* by M. Sikorski & A. Honig, 2012, No Starch Press, p. 337. © 2012 No Starch Press.

Técnicas *anti-VM* e *anti-sandbox* buscam detectar execução em ambientes virtuais ou de *sandbox*, classificadas muitas vezes como técnicas de “anti-emulação”, e alterar o comportamento do executável, retardando sua execução, limitando funcionalidades ou recusando iniciar. Os métodos mais comuns incluem verificação de artefatos de *hypervisors* (por exemplo, presença de drivers e dispositivos típicos de VM), leitura/inspeção do resultado de instruções como CPUID, checagem de interfaces de dispositivos virtuais e diversos testes de temporização para identificar diferenças entre hardware real e virtualizado. Essas abordagens tendem a ter vida útil curta, pois detecções específicas de implementações de *hypervisor* podem se tornar rapidamente obsoletas à medida que plataformas de virtualização evoluem (Blunden, 2002).

Por fim, técnicas de integridade e *anti-dump* buscam impedir modificação do executável e extração do payload da memória. Abordagens avançadas, classificadas por Ugarte-Pedrero et al. (2015) como "*Type V*" (desempacotamento incremental) e "*Type VI*" (quadros de decodificação móveis, ou *shifting-decode frames*), são projetadas especificamente para evitar o *dump* de memória. Essas técnicas desempacotam o código original apenas no

momento do acesso (por exemplo, em nível de página ou função) e, no caso do "*Type VI*", re-empacotam o código logo após sua execução, garantindo que o código completo nunca esteja visível na memória de uma só vez.

2.5 Limitações, riscos e considerações éticas

Apesar dos benefícios em termos de proteção de propriedade intelectual, o uso de *packers* apresenta limitações e riscos importantes. A quebra de assinaturas e reputação é um dos principais desafios: empacotar executáveis pode invalidar assinaturas digitais ou alterar a reputação do binário, resultando em bloqueios por mecanismos de segurança como *SmartScreen* e antivírus.

Em ambientes corporativos, políticas de execução podem impedir a execução de binários não assinados ou com propriedades atípicas. O impacto operacional também merece atenção, pois técnicas agressivas de proteção, como virtualização ou desempacotamento "*shifting-decode*" (Ugarte-Pedrero et al., 2015), aumentam a complexidade, o overhead de execução e podem gerar falsos positivos em ambientes legítimos, além de dificultar manutenção e depuração por equipes de suporte.

Outro risco relevante é o uso malicioso: as mesmas técnicas que protegem software legítimo podem ser exploradas para ocultar malware (Šťastná; Tomášek, 2016). Como o empacotamento é considerado um indicador típico de arquivos suspeitos, seu uso legítimo pode ser erroneamente penalizado. A dificuldade em distinguir, apenas por análise estática, o uso benigno do malicioso de um *packer* é um problema central. Por isso, desenvolvimentos experimentais devem ser conduzidos de forma ética, em ambientes isolados e com escopo acadêmico ou de pesquisa de segurança defensiva, documentando claramente o propósito e, quando possível, comunicando instâncias legais ou de conformidade da instituição. Limitações técnicas também devem ser consideradas, já que nem todas as aplicações são candidatas ideais ao empacotamento, incluindo drivers que exigem assinatura de kernel, módulos que expõem interfaces COM de forma estática ou sistemas que dependem de um layout binário rígido.

2.6 Recomendações para implantação e trabalhos futuros

Para implantação e trabalhos futuros, recomenda-se adotar boas práticas que aumentem a segurança e minimizem riscos. O design iterativo e testes em múltiplos ambientes são fundamentais, realizando testes exaustivos em diferentes versões do Windows, combinando políticas de segurança e ferramentas *anti-cheat* ou *anti-tamper* antes de aplicar empacotamento em produção.

A minimização da superfície de ataque do *stub* deve ser priorizada, reduzindo *strings* e dependências, evitando chaves estáticas no binário e limpando buffers sensíveis imediatamente após o uso. Sempre que possível, deve-se considerar integração com mecanismos de distribuição legítimos, preservando assinaturas digitais ou realizando re-assinatura após empacotamento para manter a reputação do software.

O foco em defesa e pesquisa deve guiar o desenvolvimento, documentando orientações com ênfase na proteção de propriedade intelectual e pesquisa acadêmica, repudiando usos ofensivos. Por fim, investigar técnicas híbridas de proteção e detecção é recomendado, unindo ofuscação com mecanismos de monitoramento e auditoria que permitam identificar manipulações ou tentativas de exploração sem degradar a experiência do usuário.

3. METODOLOGIA

A metodologia adotada para o desenvolvimento deste trabalho classifica-se como pesquisa aplicada e experimental. A abordagem é aplicada, pois visa a construção de uma solução tecnológica concreta, o packer denominado "LilithPacker", para resolver problemas práticos de proteção de software na arquitetura x64. O caráter experimental reside na validação da ferramenta através de estudos de caso quantitativos, onde foram medidas variáveis de desempenho, consumo de memória e taxa de compressão.

O desenvolvimento do projeto foi estruturado em três etapas sequenciais: (1) Análise e Levantamento Teórico, (2) Implementação do Artefato de Software e (3) Experimento de Validação.

3.1 Materiais e Ambiente de Desenvolvimento

Para a codificação e testes do *packer*, foi utilizado um ambiente controlado visando garantir a reprodutibilidade dos resultados. O sistema operacional utilizado tanto para o desenvolvimento quanto para a validação final foi o Windows 11, versão 24H2.

Como objeto de teste para a prova de conceito, selecionou-se o executável SkinnedMesh.exe. Este artefato foi compilado a partir do código-fonte *open-source* do livro "*Introduction to 3D Game Programming with DirectX 12*". A escolha deste software específico justifica-se pela sua complexidade gráfica e estrutural, sendo um cenário ideal para verificar se o processo de empacotamento comprometeria a renderização de cenas complexas em tempo real ou a estabilidade da aplicação.

3.2 Desenvolvimento do LilithPacker

A implementação do LilithPacker seguiu a arquitetura modular proposta na fundamentação técnica deste trabalho, dividida em três camadas lógicas principais:

- **Parser PE:** Foi desenvolvido um módulo de análise capaz de interpretar o cabeçalho do formato *Portable Executable* (PE) em 64 bits, extraindo informações críticas como *entry point*, tabelas de importação e seções de código.
- **Pipeline de Transformação:** Implementou-se um fluxo de operações para tratar o *payload*. Nesta etapa, aplicaram-se algoritmos de compressão para redução do tamanho do arquivo e técnicas de criptografia para ofuscação do conteúdo, protegendo-o contra análise estática.
- **Desenvolvimento do Stub:** A construção do *stub* (ou *loader*) foi realizada utilizando linguagem *Assembly* x64 e C/C++, visando minimizar dependências e permitir a manipulação direta de registradores e memória. O *stub* foi projetado para realizar a alocação de memória, resolução dinâmica de *imports*, reparação de *relocations* e a verificação de presença de depuradores através da técnica de checagem do campo *NtGlobalFlag* no PEB (*Process Environment Block*).

3.3 Procedimento de Validação e Coleta de Dados

Para validar a eficácia da solução proposta, o executável alvo (SkinnedMesh.exe) foi submetido ao processo de empacotamento pelo LilithPacker. A validação foi conduzida através de uma análise comparativa "antes e depois", focada em três indicadores chave de desempenho:

- **Integridade Funcional**: Verificação visual e lógica para assegurar que a aplicação empacotada executasse todas as suas funções originais sem falhas ou travamentos.
- **Taxa de Compressão**: Cálculo da redução percentual do tamanho do arquivo em disco após o processamento pelo pipeline de compressão.
- **Desempenho em Tempo de Execução**: Monitoramento da taxa de quadros por segundo (FPS) e do consumo de memória RAM (em Megabytes), aferidos através das ferramentas nativas de diagnóstico do sistema e dos contadores internos da própria aplicação de teste.

Os dados coletados durante os testes de execução foram registrados e comparados para demonstrar que a camada de proteção adicionada não inviabiliza o uso da aplicação em cenários reais, conforme detalhado na seção de Resultados.

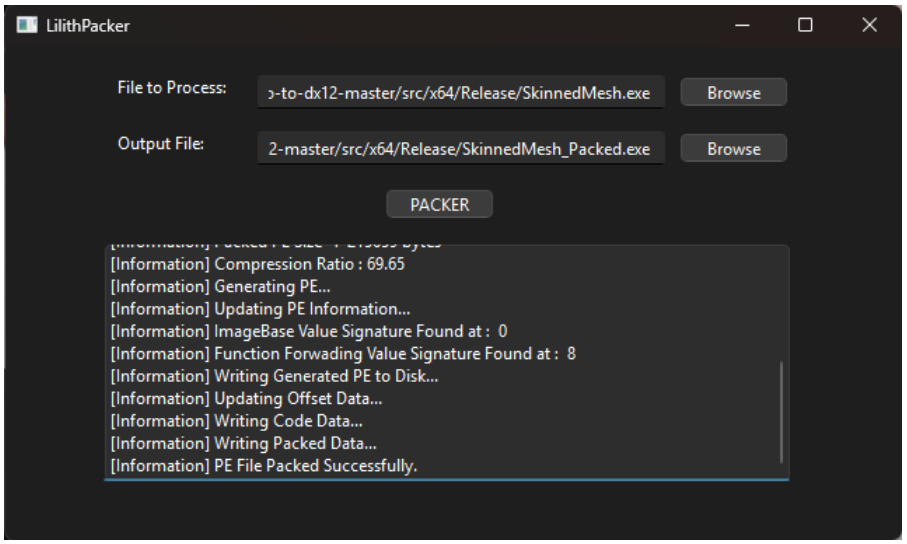
4. RESULTADOS E DISCUSSÕES

Para validar o projeto arquitetural descrito nas seções 2.2 e 2.3, foi implementada uma prova de conceito funcional, intitulada "LilithPacker". Esta ferramenta aplica as técnicas de análise, transformação de *payload* e injeção de *stub* discutidas.

Como objeto de teste, foi utilizado o executável SkinnedMesh.exe, compilado a partir do código-fonte *open-source* do livro *Introduction to 3D Game Programming with DirectX 12*. Este executável, que consiste em uma aplicação 3D que renderiza uma cena complexa em tempo real, representa um *payload* ideal devido à sua complexidade.

A Figura 3 demonstra a interface gráfica do LilithPacker processando o executável de teste e gerando o arquivo de saída (SkinnedMesh_Packed.exe).

Figura 3 – Interface do LilithPacker processando o executável de teste.

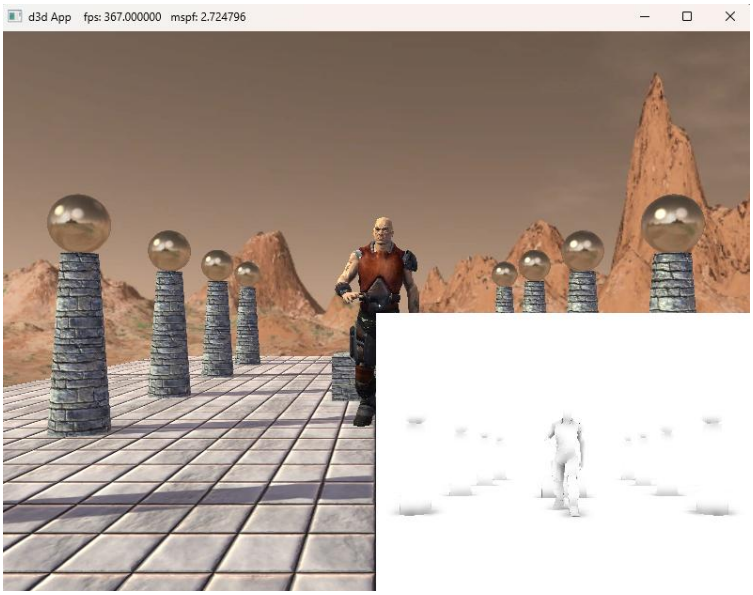


Fonte: *Elaborado pelo autor (2025)*

O log de saída da ferramenta, visível na Figura 4, fornece métricas importantes sobre o processo. Para este executável, o *packer* reportou uma taxa de compressão de 69,65% em relação ao tamanho do arquivo original, indo de 694 KB para 211 KB, mas como o código do *stub* também importa, o binário final empacotado ficou com 387 KB. Este resultado confirma a eficácia da etapa de compressão do pipeline de transformação.

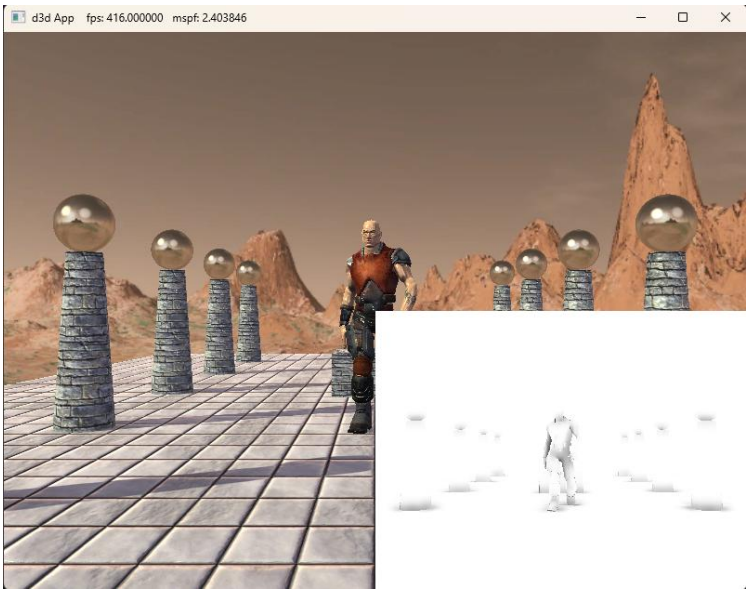
- A validação mais crítica, contudo, é a funcional. O objetivo de um *packer* legítimo é proteger o binário sem alterar seu comportamento.
- A Figura 4 exibe a aplicação original (SkinnedMesh.exe) em plena execução.
- A Figura 5 exibe a versão empacotada (SkinnedMesh_Packed.exe) executando a mesma cena.

Figura 4 – Execução da aplicação 3D original.



Fonte: *Elaborado pelo autor (2025)*

Figura 5 – Execução da aplicação 3D após empacotamento.



Fonte: *Elaborado pelo autor (2025)*

Como pode ser observado, a aplicação empacotada funciona perfeitamente, sendo visualmente idêntica à original. Isso comprova que o stub de carregamento executou com sucesso todas as suas tarefas: alocou a memória necessária, descomprimiu o payload original,

resolveu dinamicamente os imports e transferiu corretamente o controle para o ponto de entrada original.

Notavelmente, o desempenho não foi degradado. Os contadores de quadros por segundo (localizados no título da janela) visíveis nas capturas de tela indicam 367 FPS para a versão original (Figura 4) e 416 FPS para a versão empacotada (Figura 5). Embora uma análise de benchmark aprofundada esteja fora do escopo deste trabalho, os dados demonstram que o overhead do stub na inicialização é mínimo e não impacta negativamente a execução em tempo real.

Finalmente, uma análise do consumo de recursos em tempo de execução foi realizada através do Gerenciador de Tarefas do Windows, como mostra a Figura 6.

Figura 6 – Comparativo de uso de memória das aplicações.

Fonte: *Elaborado pelo autor (2025)*

A Figura 6 revela um dado interessante: a aplicação original consome 138,7 MB de memória, enquanto a versão empacotada consome 111,7 MB. Esta redução de aproximadamente 27 MB no consumo de memória pode ser atribuída à forma como o stub aloca e mapeia o payload em memória, em contraste com o mapeamento padrão realizado pelo loader do Windows para o binário original.

Estes resultados práticos validam a arquitetura proposta. O packer se mostrou funcional, capaz de ofuscar e comprimir um executável de forma eficaz, mantendo sua funcionalidade intacta e sem introduzir penalidades de desempenho ou consumo de recursos.

5. CONCLUSÃO

O presente trabalho teve como objetivo o estudo, projeto e implementação de um *packer* para executáveis Windows x64, obtendo dados por meio de pesquisa teórica e experimentação prática em ambiente controlado. A análise do formato *Portable Executable* (PE) e a aplicação de técnicas de compressão, criptografia e carregamento dinâmico permitiram demonstrar, de

forma concreta, o funcionamento interno de um *packer*. Este processo evidenciou os desafios técnicos envolvidos na manipulação de cabeçalhos, seções, *imports* e *relocations*, tarefas que o *stub* de carregamento deve executar para restaurar a semântica original do binário (Guo; Ferrie; Chiueh, 2008). O desenvolvimento reiterou a importância do domínio técnico sobre as estruturas do sistema operacional e das APIs do Windows para garantir a estabilidade e compatibilidade dos binários gerados.

Os resultados alcançados confirmaram a eficácia das técnicas de empacotamento para dificultar a análise estática e evadir a detecção baseada em assinaturas, um objetivo central tanto no uso legítimo quanto malicioso (Assis et al., 2018; Guo; Ferrie; Chiueh, 2008). Ao mesmo tempo, a implementação mostrou as limitações práticas dessas abordagens, como o aumento do overhead de CPU e latência na inicialização (Ugarte-Pedrero et al., 2015) e o risco de incompatibilidade em determinados ambientes. Verificou-se que o equilíbrio entre proteção e desempenho é essencial, e que medidas agressivas como *anti-debug*, *anti-VM* (Guo; Ferrie; Chiueh, 2008) ou técnicas de *anti-dumping* (Ugarte-Pedrero et al., 2015) devem ser aplicadas com cautela, pois podem gerar falsos positivos ou instabilidade.

As experimentações foram conduzidas de forma ética e controlada, com foco exclusivo em pesquisa acadêmica e na proteção legítima de propriedade intelectual. Esta distinção de propósito é crucial, visto que o uso de *packers* é uma característica comum tanto em softwares legítimos quanto em *malwares*, tornando a diferenciação baseada apenas em características técnicas um desafio complexo (Šťastná; Tomášek, 2016; Guo; Ferrie; Chiueh, 2008).

Conclui-se, portanto, que o projeto cumpriu seu propósito ao apresentar uma solução funcional, modular e segura de *packer* para a arquitetura Windows x64, além de contribuir para o avanço do conhecimento técnico na área de segurança de software. Recomenda-se, para trabalhos futuros, o aprimoramento do desempenho do *stub*, a integração com mecanismos de assinatura digital e o estudo de técnicas híbridas de proteção, considerando a contínua evolução na complexidade estrutural de protetores (Ugarte-Pedrero et al., 2015). Dessa maneira, o trabalho reforça a importância da pesquisa responsável e do uso ético de ferramentas de ofuscação, conciliando inovação tecnológica com rigor científico e compromisso com a segurança da informação.

6. TRABALHOS FUTUROS

A partir da prova de conceito funcional “LilithPacker”, desenvolvida e validada neste estudo, abrem-se diversas possibilidades de aprofundamento técnico e científico voltadas ao aprimoramento da ferramenta e à ampliação de seu uso prático. O projeto, ao demonstrar a viabilidade de um *packer* customizado para executáveis Windows x64, estabelece uma base sólida sobre a qual é possível explorar melhorias voltadas à robustez, eficácia, segurança e compatibilidade da solução.

Um dos eixos mais promissores para trabalhos futuros diz respeito à integração com mecanismos de assinatura digital. Atualmente, uma das principais limitações de ferramentas de empacotamento é a quebra das assinaturas legítimas presentes nos binários originais, o que compromete a confiança do sistema operacional e dos usuários, além de dificultar a distribuição em ambientes corporativos ou lojas oficiais. Uma abordagem de pesquisa relevante seria a implementação de um processo automatizado de re-assinatura digital após o empacotamento, utilizando certificados válidos ou cadeias de confiança reconhecidas. Esse aprimoramento permitiria que o *packer* se integrasse a fluxos de desenvolvimento e distribuição legítimos, preservando a reputação do software, sua autenticidade e conformidade com políticas de segurança. Além disso, seria possível investigar a compatibilidade da ferramenta com diferentes algoritmos de assinatura (como SHA-256 e SHA-3) e métodos de *timestamping* para garantir a validade temporal das assinaturas.

Outro campo fundamental de expansão envolve a ampliação dos testes de compatibilidade e robustez. Embora a versão atual do LilithPacker tenha sido validada em um ambiente controlado (Windows 11 Versão 24H2), um estudo mais abrangente deve incluir testes em diferentes versões do Windows, avaliando o comportamento do executável empacotado em contextos reais de uso. Também se recomenda analisar a interação do *packer* com diversas soluções de segurança, incluindo antivírus, mecanismos *anti-tamper, anti-debug* e sistemas *anti-cheat*, de modo a identificar e mitigar eventuais conflitos ou falsos positivos. Esses testes poderiam ser conduzidos de forma sistemática, com o apoio de ferramentas automatizadas de análise dinâmica, permitindo uma validação mais ampla e consistente da compatibilidade da ferramenta.

Adicionalmente, outras frentes de investigação podem ser consideradas, como a otimização do desempenho do *stub* de desempacotamento, buscando reduzir o tempo de inicialização do executável empacotado e o consumo de memória durante o processo de extração. Também é possível explorar novas técnicas de compressão e criptografia que

equilibrem eficiência e segurança, bem como a incorporação de técnicas anti-análise mais sofisticadas, voltadas a aumentar a resistência do empacotador frente a ferramentas de engenharia reversa.

REFERÊNCIAS

ASSIS, C. R. O. et al. **A Comparative Analysis of Classifiers in the Recognition of Packed Executables**. Uberlândia: Federal University of Uberlândia, 2021.

BLUNDEN, Bill. **Virtual Machine Design and Implementation in C/C++**. 1. ed. Burlington: Wordware Publishing, Inc., 2002. 784 p. v. 1. ISBN 9781556229039.

FERRIE, Peter. **The Ultimate Anti-Debug Reference**. pferrie.epizy.com, 2011. Disponível em: https://pferrie.epizy.com/papers/antidebug.pdf . Acesso em: 25 out. 2025.

GUO, F.; FERRIE, P.; CHIUEH, T. A Study of the Packer Problem and Its Solutions. In: LIPPMANN, R.; KIRDA, E.; TRACHTENBERG, A. (Eds.). **Recent Advances in Intrusion Detection (RAID 2008)**. Berlin, Heidelberg: Springer-Verlag, 2008.

SIKORSKI, Michael; HONIG, Andrew. **Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software**. 1. ed. San Francisco: No Starch Press, 2012. 800 p. v. 1. ISBN 9781593272906.

ŠŤASTNÁ, J.; TOMÁŠEK, M. **The Problem of Malware Packing and Its Occurrence in Harmless Software.** Acta Electrotechnica et Informatica, v. 16, n. 3, 2016.

UGARTE-PEDRERO, X. et al. **SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers**. In: IEEE Symposium on Security and Privacy, 2015.

Agradecimentos

Primeiramente agradeço meu orientador, professor Álvaro, pela orientação constante, paciência e dedicação ao longo de todas as etapas deste trabalho. Sua disponibilidade em esclarecer dúvidas, revisar cada parte do estudo e oferecer conselhos técnicos e teóricos foi fundamental para o desenvolvimento deste artigo. Sua experiência e comprometimento serviram como exemplo de rigor acadêmico, tornando este processo não apenas um aprendizado técnico, mas também uma experiência pessoal de crescimento intelectual.

Agradeço também à comunidade de tecnologia e pesquisa em segurança da informação, que, por meio de fóruns, repositórios, publicações e ferramentas de código aberto, contribui

continuamente para a disseminação do conhecimento e o avanço coletivo da área. O acesso a recursos livres de aprendizado, documentações detalhadas e discussões abertas tornou possível explorar, compreender e aplicar diversos conceitos abordados neste estudo.

Por fim, deixo meu reconhecimento a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho, colegas, amigos e familiares, pelo apoio, incentivo e compreensão durante as fases de pesquisa, desenvolvimento e redação.