UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

João Vitor Maia Neves Cordeiro

Desenvolvimento de uma técnica de esteganografia explorando arquivos binários de código compilados para RISC-V

Florianópolis
2023

João Vitor Maia Neves Cordeiro

Desenvolvimento de uma técnica de esteganografia explorando arquivos binários de código compilados para RISC-V

Trabalho de Conclusão de Curso do Curso de Graduação em Ciência da Computação do Campus Florianópolis da Universidade Federal de Santa Catarina para a obtenção do título de bacharel em Ciência da Computação.
Orientador: Prof. Jean Everson Martina, Dr.

Florianópolis
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Cordeiro, João Vitor Maia Neves
Desenvolvimento de uma técnica de esteganografia explorando arquivos binários de código compilados para RISC V / João Vitor Maia Neves Cordeiro ; orientador, Jean Everson Martina, 2023.
71 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2023.

Inclui referências.

1. Ciências da Computação. 2. Segurança da Informação. 3. Esteganografia. 4. RISC-V. I. Martina, Jean Everson. II. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. III. Título.

João Vitor Maia Neves Cordeiro

Desenvolvimento de uma técnica de esteganografia explorando arquivos binários de código compilados para RISC-V

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "bacharel em Ciência da Computação" e aprovado em sua forma final pelo Curso de Graduação em Ciência da Computação.

Florianópolis, 29 de Novembro de 2023.

Prof. Lúcia Helena Martins Pacheco, Dra.
Coordenadora do Curso

Banca Examinadora:

Prof. Jean Everson Martina, Dr.
Orientador

Prof. Alexandre Augusto Giron, Dr.
Avaliador

Prof. Frederico Schardong, Me.
Avaliador

Este trabalho é dedicado a minha mãe e todos os sacrifícios que ela fez para que eu pudesse chegar até aqui.

AGRADECIMENTOS

Deixo meus profundos agradecimentos à minha revisora preferida, Maria Eduarda da Luz, por não deixar meus inúmeros erros passarem para o texto final; ao meu orientador Jean Everson Martina que acreditou e confiou na minha proposta inicial; ao meu professor e amigo Rodrigo Ramos Nogueira pelas companhias nas madrugadas de trabalho; e por fim a todos os meus amigos que me ajudaram a construir quem sou hoje.

"Apparently neutral s protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on by-products, ejecting suets and vegetable oils."

(Espião Desconhecido, 19??)

RESUMO

A esteganografia trabalha embutindo informação dentro de outra e escondendo essa transformação ao olhar de um observador ingênuo, de forma que não seja possível distinguir a mídia original do resultado transformado. Apesar do campo já possuir técnicas consolidadas para lidar com imagens, áudios e outros tipos de mídia, quando trata-se de arquivos de código binário a literatura e o ferramental atual são escassos. Dado esse contexto, esse trabalho apresenta o desenvolvimento de um algoritmo capaz de inserir informações dentro de um arquivo executável compilado para a arquitetura RISC-V sem alterar a semântica do programa fonte, bem como sua implementação em uma aplicação com interface em linha de comando, podendo ser utilizado para finalidades como assinatura digital de software, proteção de direitos autorais e comunicação oculta. Os resultados obtidos indicam que arquiteturas RISC produzem código de máquina capaz de abrigar uma quantidade reduzida de informação oculta quando comparadas com arquiteturas CISC, estreitando o alcance de aplicações de esteganografia para aquelas que não necessitam transmitir uma quantidade de dados elevada.

Palavras-chave: Esteganografia. Segurança da Informação. Ocultamento de dados. RISC-V.

ABSTRACT

Stegranography is the craft of embedding information inside another source of data, hiding it from an external observer. Despite the extensive research on techniques built for image, audio and text steganography, when it comes to using executable files as cover objects the academic productions and tools are scarce. This article presents a method capable of embedding information inside an executable file compiled for the RISC-V architecture, without altering the semantics of the source program, and implements it as a command line application that may be used to a wide range of purposes, like digital signature, copyright protection and hidden communication. The results suggests that RISC architectures produces machine code capable of hiding a reduced amount of information when compared to CISC architecture, narrowing the scope of steganographic applications for those that do not require a high encoding rate.

Keywords: Steganography. Information Security. Information Hiding. RISC-V.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

AES	Advanced Encryption Standard
ARM	Advanced RISC Machine
ASCII	American Standard Code for Information Interchange
CISC	Complex Instruction Set Computer
ELF	*Executable and Linkable File*
EPOS	Embedded Parallel Operating System
HMAC	Hash-based Message Authentication Code
ISA	Instruction Set Architecture
PBKDF2	Password-Based Key Derivation Function 2
PE	*Portable Executable*
POSIX	Portable Operating System Interface
RISC	Reduced Instruction Set Computer
ROP	Return Oriented Programming
SGBD	Sistema Gerenciador de Banco de Dados
SO	Sistema Operacional

SUMÁRIO

1 INTRODUÇÃO

Esteganografia é a nomenclatura utilizada para um conjunto de técnicas, inicialmente desenvolvidas para espionagem, que atuam escondendo uma informação dentro de uma outra mídia, de modo que a mídia criada após a manipulação seja visualmente indistinguível da original para um observador externo.

A crescente disseminação de computadores a partir do século 20 recuperou diversas técnicas analógicas de espionagem que predatam a área da computação, adicionando o poder de processamento digital para aprimorar a efetividade e as aplicações de tais técnicas. Notoriamente temos a criptografia como a principal expoente dessa transformação, estando presente na maior parte das aplicações modernas. Apesar de ser o maior caso a criptografia não foi a única área a passar por isso, as técnicas de esteganografia também foram afetadas pela introdução do fator digital (JOHNSON; JAJODIA, 1998).

Representações digitais de mídias diversas permitem que a informação seja processada e escondida de formas que uma mídia física, por sua natureza, não conseguiria suportar, fornecendo uma gama maior de técnicas e aplicações esteganográficas. Além disso, o meio digital permite a combinação das técnicas mais modernas de criptografia para esconder uma mensagem cifrada.

As novas técnicas esteganográficas desenvolvidas no meio digital circulam por diversas áreas da computação, nos últimos anos destaca-se o uso no ramo de marcas d'água digitais, onde são inseridas no arquivo informações para ajudar na verificação de sua integridade, autenticidade e dados autorais. Nessas aplicações, os dados embutidos não são necessariamente secretos portanto a invisibilidade nesses casos é apenas uma propriedade opcional do algoritmo e outros fatores surgem como prioridade. Uma dessas novas características que ganham destaque é a capacidade de resistir a ataques de distorção do arquivo, mantendo os dados embutidos seguros de um atacante que queira danificar a integridade do arquivo fonte (EVSUTIN; MELMAN; MESHCHERYAKOV, 2020).

1.1 MOTIVAÇÃO

Apesar do comum ser utilizar técnicas esteganográficas em imagens, áudios e outros formatos de mídia de uso geral, os mesmos conceitos podem ser aplicados para qualquer formato de arquivo que contenha informação, ajustando ou criando novos algoritmos. Uma aplicação menos explorada atualmente é a esteganografia em arquivos de código binário, que pode ser utilizada para transmissão de mensagens secretas, marcas d'água digitais em software, verificação de integridade e autoria a partir de uma assinatura e até mesmo inclusão de trechos de código alternativo, maliciosos ou não, dentro do programa principal (MAHONEY *et al.*, 2018).

Os softwares que consumimos diariamente muitas vezes são compilados em arquivos binários que codificam as linhas de código fonte em instruções binárias compreendidas

pelo processador alvo. Esses arquivos compilados também possuem extensões e formatos, sendo os principais formatos utilizados o *Executable and Linkable File* (ELF) e *Portable Executable* (PE), sendo que o último possui diversas versões.

Trabalhos passados já obtiveram resultados no desenvolvimento e validação de técnicas esteganográficas em arquivos binários, notóriamente o projeto Hydan foi um dos precursores ao criar uma técnica para a arquitetura *x86* capaz de incorporar bits de informação em um arquivo na proporção de $\frac{1}{110}$. Em comparação, técnicas que se utilizam de mídias mais suscetíveis como imagens JPEG podem atingir uma eficiência de $\frac{1}{17}$, o que contribui para a aplicação de técnicas de obfuscação dentro do espaço de busca (EL-KHALIL; KEROMYTIS, 2004).

Em seguida novas pesquisas foram realizadas em torno da ideia inicial do Hydan, buscando aumentar a taxa de incorporação das técnicas empregadas utilizando-se de heurísticas mais efetivas (ANCKAERT *et al.*, 2004), expondo vulnerabilidades contra ataques baseados no espaço de busca pequeno que o algoritmo gera (WRIGHT, 2008). Entretanto, as pesquisas da área estão limitadas em dois sentidos: a falta de novas propostas de algoritmos que ampliem a capacidade de codificação por arquivo e a preocupação de tornar as técnicas já existentes disponíveis ao longo de mais plataformas, visto que os trabalhos já desenvolvidos costumam focar exclusivamente na arquitetura x86.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Desenvolver um algoritmo para a utilização de esteganografia em arquivos binários de código fonte em arquiteturas alternativas à x86, visando ampliar a aplicabilidade de técnicas esteganográficas em diferentes plataformas.

1.2.2 Objetivos Específicos

- Pesquisar o estado da arte dos algoritmos de esteganografia utilizando arquivos de código compilado como mídia.
- Pesquisar e documentar brechas na arquitetura alvo que permitam a codificação de mensagens dentro de um arquivo compilado.
- Desenvolver um algoritmo que explore as brechas encontradas para aplicar técnicas esteganográficas em código da arquitetura alvo.
- Desenvolver uma aplicação que implemente o algoritmo desenvolvido.
- Disponibilizar a aplicação de forma pública para uso da comunidade e evolução do campo.

1.3 ESTRUTURA DO TRABALHO

Além do presente capítulo introdutório, este trabalho conta com mais cinco capítulos, explicados a seguir.

No capítulo 2 são apresentados conceitos gerais para a compreensão da temática do trabalho e seus materiais relacionados. São superficialmente explicados noções e definições basilares sobre esteganografia e ofuscação, contendo um breve histórico da formação e avanço das ideias trabalhadas. Também neste capítulo é apresentada uma introdução sobre arquivos executáveis de código e a arquitetura RISC-V escolhida como alvo do estudo. Ao final é exposta a revisão bibliográfica utilizada como fundamentação para o trabalho.

Em seguida está detalhado o desenvolvimento do trabalho, dividido em duas partes. A primeira parte presente no Capítulo 3 discorre sobre a elaboração teórica do algoritmo proposto, detalhando as escolhas de projeto e aplicando uma análise de complexidade, robustez e eficiência de codificação de informação do método. Já a segunda parte exposta no Capítulo 4 detalha a implementação do algoritmo juntamente com uma aplicação com interface de linha de comando para a utilização do algoritmo.

No Capítulo 5 são discutidos os resultados da implementação, realizando uma análise quantitativa em comparação com técnicas aplicadas em outras arquiteturas por trabalhos correlatos e uma análise qualitativa a partir de modelos de validação de corretude de métodos esteganográficos. Por fim, no Capítulo 6 o conhecimento adquirido durante o estudo é sumarizado a fim de demonstrar a conclusão dos objetivos propostos, assim como possibilidades de trabalhos futuros a serem desenvolvidos orbitando o presente trabalho.

2 CONCEITUAÇÃO E REVISÃO BIBLIOGRÁFICA

Ao longo deste capítulo serão introduzidos conceitos necessários para a compreensão do algoritmo proposto como objetivo, com uma pesquisa calcada tanto em materiais e documentos históricos quanto produções acadêmicas recentes de maior relevância.

2.1 CONCEITOS GERAIS

Sabendo que o objetivo principal é inovar na adaptação de algoritmos de esteganografia para arquiteturas além da *x86*, essa seção expõe os conceitos básicos relacionados aos principais métodos de esteganografia, conceitos históricos e trabalhos pioneiros que abordam a problemática.

2.1.1 Estegranografia

2.1.1.1 Definição e Histórico

Etimologicamente a palavra esteganografia deriva do grego, *steganographia*, que significa de forma literal "escrita escondida". Historicamente a palavra já foi utilizada para definir um conjunto bem abrangente de técnicas ao decorrer dos seículos, iniciando-se com ocultação de símbolos em placas de hieróglifos egípicios, passando por técnicas de alteração química em tintas que só seriam exibidas em determinadas circustâncias durante a Segunda Guerra Mundial e chegando nos dias atuais com o uso de algoritmos computacionais para ocultação de mensagens em arquivos digitais.

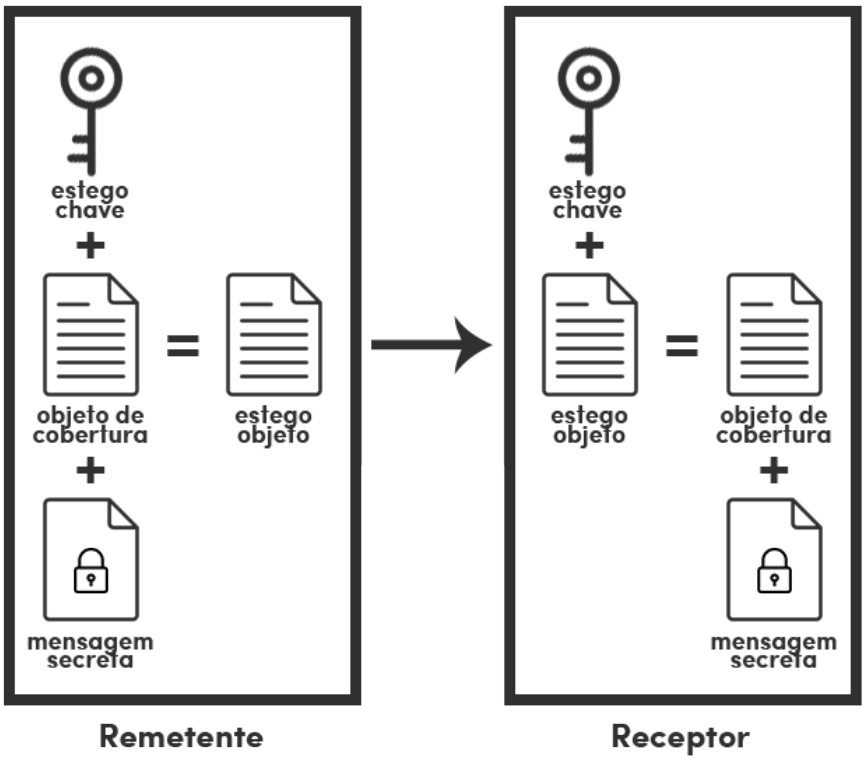
Conceitualmente, a primeira definição registrada vem do polímata Johannes Trithemius na obra *Steganographia*, escrita em 1499 e publicada apenas após sua morte em 1606 em três partes. No subtítulo do tratado, originalmente em latim e traduzido por Judge, o autor alemão introduz esteganografia como a arte pela qual a escrita é escondida e requer recuperação pela mente humana (JUDGE, 2001).

A publicação em si é uma demonstração do uso da escrita escondida, pois inicialmente foi reconhecida como um tratado sobre demonologia de teor teológico, e só após um estudo mais aprofundado foram encontradas mensagens escondidas com metodologia sofisticadas nos dois primeiros livros. O terceiro livro só foi decifrado em 1998 por Thomas Ernst (ERNST, 1998) e Jim Reeds (REEDS, 1998) que trabalharam sem conhecimento da pesquisa um do outro.

Com a evolução da área e do campo científico como um todo, se fez necessário definir esteganografia de forma menos esotérica do que a usada por Trithemius. De forma objetiva, esteganografia é o uso de qualquer processo capaz de ocultar a existência de uma mensagem em um objeto externo, tornando essa mensagem opaca para qualquer observador que não tenha o conhecimento prévio da aplicação da técnica (JUDGE, 2001).

Segundo Katzenbeisser (1999) podemos generalizar grande parte dos métodos esteganográficos em um princípio comum: o remetente deseja compartilhar uma mensagem secreta (m) com um receptor pré informado, para isso ele escolhe um **objeto de cobertura** (c) que possa ser visto como uma mensagem pública. Então, utilizando-se de algum método específico, o remetente insere m dentro de c. Pode-se ainda utilizar uma **estego-chave** (k) para controlar o processo de inserção de m. O resultado desse processo é chamado de **estego-objeto**, e deve ter uma estrutura que permita a extração de m desde que k (caso exista) seja conhecido, sem que seja necessário conhecer o c original. Uma mesma cobertura não deve ser utilizada duas vezes, já que um atacante em posse de duas versões de c pode utilizar as divergências entre elas para detectar a existência de uma mensagem e possivelmente recuperar m (KATZENBEISSER, 1999). Tal esquema está ilustrado na Figura 1.

Figura 1 – Diagrama geral de um método esteganográfico.



Fonte: Imagem produzida pelo Autor.

2.1.1.2 Tipos de Esteganografia

Como citado anteriormente, a chave k é opcional, sendo que sua presença e tipo podem classificar um método de esteganografia em 3 classes: esteganografia pura, esteganografia de chave privada e esteganografia de chave pública. Caso não exista uma chave,

então temos uma esteganografia pura onde não há nenhuma troca de informações secretas anterior à comunicação. Apesar de possuírem implementações normalmente mais simples, métodos de esteganografia pura ferem o segundo princípio de Kerckhoffs (KERCKHOFFS, 1883), que aconselha que a segurança de um sistema de cifragem seja baseada em uma chave e não no algoritmo de cifragem, dessa forma caso o funcionamento do sistema seja conhecido pelo atacante, ele ainda não é capaz de quebra-lo sem a chave. Por consequência disso, o recomendado é que se utilizem os métodos baseados em chaves.

Na esteganografia de chave privada, é necessário que as duas partes compartilhem uma única chave que será utilizada durante o processo de inserção da mensagem secreta, de forma impossibilitar que qualquer atacante sem possa da chave seja capaz de decifrar a mensagem mesmo que tenha conhecimento do algoritmo. Essa classe possui conceitos que são análogos à criptografia simétrica, herdando portanto seus contratempos, sendo o principal deles a necessidade de um canal seguro para que a chave privada seja compartilhada antes do início da comunicação (ANDERSON, 1996).

A terceira classe de métodos é derivada da criptografia assimétrica, exigindo de um par de chaves, uma pública e uma privada. Durante o processo de inserção dos dados no objeto de cobertura, a chave pública é utilizada, enquanto para a recuperação da mensagem original é usada a chave privada. Pode-se inclusive construir um método de esteganografia de chaves públicas em cima de uma base de um sistema criptográfico de chaves públicas (ANDERSON, 1996).

2.1.1.3 Aplicações Modernas

Em tempos anteriores à popularização de computadores a esteganografia possuía um leque de aplicações fortemente confinado à transmissão de mensagens secretas entre dois correspondentes. A razão disso passa primeiramente pela falta de eficiência nos métodos utilizados, e em seguida pelo fato de que as aplicações modernas que conhecemos hoje sequer eram um assunto existente na sociedade da época.

O campo de marcas d'água digitais surge com a necessidade de inserir e verificar provas de autoria e propriedade de recursos digitais, e avança conforme as legislações específicas de grandes países ou blocos são sancionadas. Entretanto, nessa área os requisitos para a utilização de um algoritmo diferem dos habituais, dado que existe a necessidade de impedir que a marca seja retirada ou corrompida por transformações no arquivo. Além disso, a invisibilidade completa da mensagem nem sempre é indispensável e podemos ter marcas d'água visíveis para um observador externo. Mesmo com essas ressalvas, é notório que os últimos anos de evolução das técnicas de esteganografia foram fortemente voltados para aplicações na área de marcas d´água digitais, por seu apelo comercial latente (EVSUTIN; MELMAN; MESHCHERYAKOV, 2020).

Apesar da natureza distinta entre as duas áreas, grande parte dos algoritmos utilizados para esteganografia podem ser transpostos para a aplicação de marcas d´água,

visto que possuem a robustez necessária para impedir a mensagem de ser violada e que a ocultação da mesma não é um impeditivo (EVSUTIN; MELMAN; MESHCHERYAKOV, 2020). E, de fato, possuímos exemplos de usos extensivos como os micropontos invisíveis a olho nu adicionados por impressoras comerciais em suas impressões (EMBAR; MCHUGH; WESSELMAN, 2014) e as câmeras digitais com capacidade de adicionar uma marca invisível em cada foto que tiram (BLYTHE; FRIDRICH, 2009).

2.1.1.4 Problema dos Prisioneiros

O problema dos prisioneiros é um modelo utilizado para para descrever comunicação subliminar a partir de um canal monitorado. No problema os dois prisioneiros Armando e Bruna estão confinados em celas diferentes e Chico, o carcereiro, inspeciona toda a comunicação entre eles, descartando qualquer tipo de mensagem criptografada ou de conteúdo suspeito, forçando os dois a traçarem seu plano de fuga utilizando apenas comunicação invisível (SIMMONS, 1984). Podemos assumir que em um momento anterior à prisão, os dois prisioneiros combinaram uma chave a ser utilizada para algum tipo de comunicação e que o carcereiro atua de forma passiva apenas validando as mensagens sem alterá-las. Além disso, assumimos que o canal de comunicação é conhecido por Chico, para obedecer ao segundo princípio de Kerckhoffs.

Dessa forma podemos utilizar o diagrama exposto na Figura 1 para descrever uma troca de mensagens dentro do problema dos prisioneiros. Em uma cela, Armando escolhe uma mensagem inocente que será utilizada como objeto de cobertura e insere a mensagem secreta processada a partir da estego chave, gerando um estego objeto que é passado pelo canal de comunicação monitorado por Chico, e em outra cela, Bruna recebe o estego objeto e reconstrói a mensagem secreta a partir da chave combinada. Caso essa comunicação seja feita sem que o carcereiro consiga detectar a existência de uma mensagem no estego objeto podemos dizer que ela foi bem sucedida e invisível. Ao longo desse trabalho iremos utilizar esse problema de forma a buscar uma solução satisfatória para ele.

2.1.2 Arquivos Executáveis

A maioria das pessoas programadoras atualmente escreve código em linguagens de alto nível que são muito próximas de línguas humanas, enquanto computadores modernos interpretam sequências de instruções representadas em base binária. Isso só é possível graças a compiladores, softwares que transformam construções feitas em linguagens de alto nível para o que chamamos de linguagem de máquina (DAVID A. PATTERSON, 2014).

O processo clássico de compilação usualmente toma como parâmetros de inicialização um código fonte escrito em uma linguagem de alto nível e o traduz para linguagem de máquina que posteriormente pode ser salvo em um arquivo, chamado de arquivo executável (AHO *et al.*, 2006). Esse arquivo é capaz de ser carregado para o processador

e executar as instruções dadas pelo programa. Esses arquivos podem possuir estruturas e formatos diferentes, dependendo do sistema operacional e do processador alvos que foram usados. Atualmente os dois formatos mais utilizados são o ELF e PE, utilizados respectivamente pelos sistemas operacionais Linux (LINUX STANDARD BASE..., s.d.) e Windows (MICROSOFT..., s.d.). Apesar de possuirem diferenças, ambos os formatos citados compartilham características. Além do código já devidamente representado em linguagem de máquina, os arquivos carregam um cabeçalho com metadados referentes ao programa.

2.1.2.1 ELF (*Executable and Linkable Format*)

O formato ELF foi desenvolvido inicialmente pela Unix System Laboratories em 1988, ganhando notoriedade e implementações em outros sistemas, até que em 1999 foi adotado como padrão pelo sistema Unix e seus derivados. Por ser facilmente generalizado para diversos tipos de sistemas e possuir suporte para a maioria das funcionalidades utilizadas pela indústria atualmente, continua até hoje figurando entre os formatos mais relevantes para arquivos executáveis.

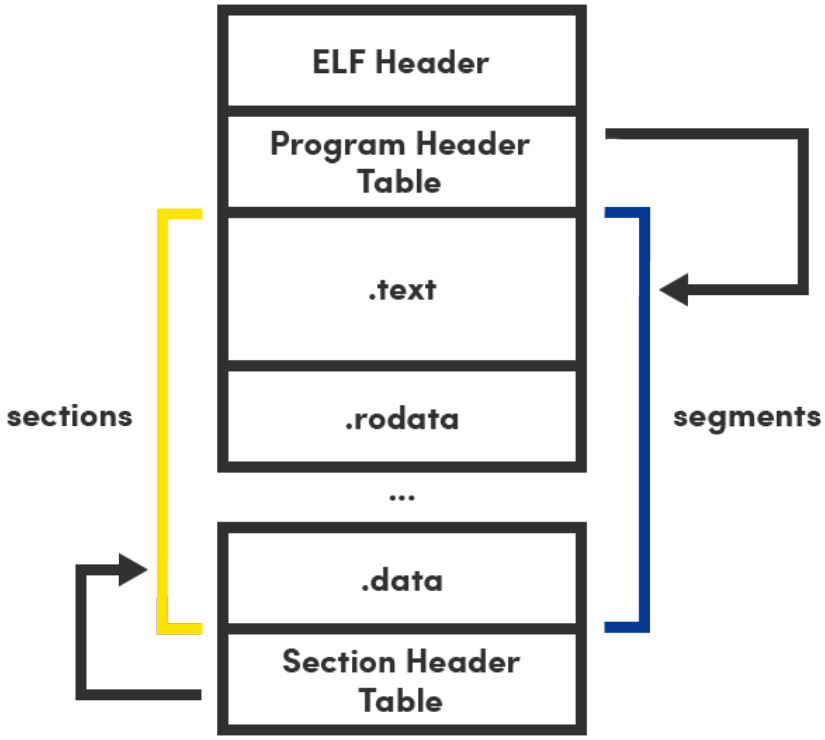
Estruturalmente, um arquivo ELF é dividido em duas estruturas principais: segmentos e seções. Os segmentos contém informações úteis para o programa durante seu tempo de execução, como o local e valores de constantes definidas em código e as próprias instruções do programa. As seções possuem dados utilizados em tempo de ligação ou para *debugging* como símbolos gerados no processo de compilação e endereços de realocação de memória (TOOL INTERFACE STANDARD (TIS)..., 1995).

Para organizar essas informações, é utilizado um esquema de cabeçalhos contendo metadados e posições dos segmentos e seções. Logo no início do arquivo está o *ELF Header* detalhando informações como arquitetura alvo, tipo de arquivo e posições para os próximos cabeçalhos. Em seguida começam os *Program Headers* que definem como estão dispostos os segmentos no arquivo, os próprios segmentos e seções com seus conteúdos e por fim os *Section Headers* contendo as informações das seções. A estrutura típica de um executável contendo código e dados pode ser vista na Figura 2, notando que é comum essa sobreposição entre secções e segmentos na memória (TOOL INTERFACE STANDARD (TIS)..., 1995).

2.1.3 Arquitetura RISC-V

Assim como nem todos os humanos compreendem a mesma língua, nem todas as máquinas sabem interpretar a mesma linguagem. Existem diversas formas de representar a linguagem de máquina desenvolvidas ao passar das décadas por diferentes fabricantes de computadores, cada máquina possuindo seu próprio conjunto de instruções que é capaz de executar. Esse conjunto é comumente denominado Instruction Set Architecture (ISA) e todas as máquinas que compreendem um mesmo conjunto são ditas da mesma arquitetura.

Figura 2 – Estrutura de um arquivo ELF contendo dados e código.



Autor: Imagem produzida pelo Autor.

Atualmente duas grandes filosofias de construção de arquiteturas disputam espaço no mercado, Complex Instruction Set Computer (CISC) e Reduced Instruction Set Computer (RISC), tendo como mote principal a complexidade do seu conjunto de instruções. Enquanto máquinas CISC possuem um conjunto de instruções maior e essas instruções também contém um grau de complexidade mais elevado, as máquinas RISC possuem poucas operações e buscam realizar as operações de forma fragmentada em uma quantidade maior de passos simples (PATTERSON; SEQUIN, 1981).

Em consequência dos princípios básicos de cada tipo de máquina, é possível inferir características que serão comuns a todas as arquiteturas de cada classe. Máquinas CISC costumam gerar códigos menores, pois aglutinam diversas operações em uma só instrução, enquanto nas implementações RISC temos o oposto. Entretanto, apesar da quantidade total de instruções geradas para um mesmo código na máquina RISC ser maior, cada instrução possui uma execução mais rápida devido a sua simplicidade, retomando de volta a performance perdida por executar um número maior de instruções (PATTERSON; SEQUIN, 1981).

Nos dias atuais a mais popular representação de uma arquitetura CISC é a x86, desenvolvida inicialmente pela Intel em 1978 como uma extensão da arquitetura 8080 (ASANOVIĆ; PATTERSON, 2014). Apesar de ser coloquialmente referida como uma

arquitetura, é na verdade um conjunto que abarca diversas ISAs desenvolvidas ao longo das décadas. Para máquinas RISC podem ser citadas as arquiteturas Advanced RISC Machine (ARM) e RISC-V, sendo que a última possui esse nome por ser a quinta geração de arquiteturas RISC a ser desenvolvida pela Universidade da Califórnia (PATTERSON; HENNESSY, 2017).

A arquitetura RISC-V é uma ISA desenvolvida de forma aberta e sem cobrança de *royalties*, originalmente pelo departamento de engenharia elétrica e ciências da computação da Universidade da Califórnia, campus Berkeley. Entretanto, desde o início em 2010 até o presente, muitos contribuidores externos, independentes ou em contribuições corporativas, participaram do projeto. Diferentemente do usual para projetos acadêmicos, o RISC-V busca não apenas ser um objeto de estudo teórico, mas também ter viabilidade para projetos reais na indústria (ASANOVIĆ; PATTERSON, 2014).

O RISC-V possui um conjunto de apenas 47 instruções obrigatórias, complementadas por extensões padronizadas de propósitos gerais que adicionam funcionalidades como multiplicação, operações atômicas e aritmética de ponto flutuante. Todas as operações são realizadas exclusivamente em registradores, utilizando o conceito de *load/store machine*, para operar sobre um valor em memória primeiro é necessário uma instrução para carregá-lo em um registrador, em seguida é realizada a operação em uma segunda instrução, e por fim uma última instrução retorna o valor para a memória (WATERMAN; LEE *et al.*, 2014).

2.2 TRABALHOS RELACIONADOS

Apesar de não ser um assunto extensamente explorado, a esteganografia utilizando código executável já foi proposta e estudada por outros autores ao longo dos últimas décadas. O projeto Hydan utiliza instruções semânticamente equivalentes como decisores binários para inserir dados dentro de um código executável para a arquitetura x86. Entretanto, a razão de codificação de em média 1 bit de mensagem para cada 100 bits do objeto de cobertura encontrada é insatisfatória para alguns casos de uso (EL-KHALIL; KEROMYTIS, 2004). No mesmo ano, Anckaert *et al* (2004) expandem o conceito inicial do Hydan, realizando um estudo sobre a redundância em programas da arquitetura x86 e formulando critérios de substituição claros. Além disso, esse trabalho também traz um *framework* para análise de detectabilidade dos estego-objetos gerados.

O *whitepaper* publicado por Wright (2008) desenvolve em cima de um tópico já citado pelos autores originais do Hydan: a fragilidade do trabalho em relação a detectabilidade. O método aplicado pelos pesquisadores não leva em consideração a distribuição estatística das instruções em cada plataforma, permitindo que uma análise seja feita buscando anomalias, comparando instruções contidas no programa alvo com a distribuição normalmente encontrada em programas compilados para a mesma plataforma (WRIGHT, 2008).

Outra abordagem foi tomada por Shin *et al* (2008), que desenvolve um método para inserção de dados em arquivos do formato PE, adicionando os dados na seção *.text* do binário e tomando providências de realocação caso seja necessário. Apesar desse método possuir uma flexibilidade maior e capacidade de codificação virtualmente infinita, ele altera o tamanho do arquivo em disco, portanto pode ser detectado por um observador externo em posse do objeto de cobertura, invalidando essa técnica para aplicações em que o arquivo original seja de alguma forma público. De forma complementar, um trabalho posterior propõe um método similar cobrindo as mesmas plataformas e adicionando uma camada de criptografia (ZAIDAN; BAHAA; OTHMAN, 2009).

Além de métodos diferentes para a inserção dos dados no objeto de cobertura, a natureza dos próprios dados é espaço de pesquisas. Lu, Xiong e Gao (2014) elaboraram a ideia de utilizar esteganografia não só para esconder uma mensagem dentro de um código fonte, mas sim para esconder parte do código fonte dentro de si mesmo. Os autores utilizam Return Oriented Programming (ROP) em combinação com esteganografia para que certos conjuntos de instrução sejam invisíveis à analisadores estáticos de código mas ainda sejam executadas em tempo de execução. Como prova de conceito é apresentada a ferramenta RopSteg que utiliza uma versão modificada do algoritmo de *Galileo* para gerar segmentos de código contendo estruturas de retorno que não estavam no binário original. Entretanto, apesar de conseguirem êxito para fazer as alterações no arquivo, devido à ROP ser frequentemente associada com malwares, os autores alertam que existe a possibilidade do arquivo gerado ser marcado como malicioso por sistemas de detecção (LU; XIONG; GAO, 2014).

Por fim, existem ainda implementações não acadêmicas porém relevantes, utilizando propostas levantadas pelos autores anteriores e que de certa formam fornecem alguma validação da viabilidade material desses métodos. A ferramenta *steg86* apropria-se da ideia inicial dos autores do Hydan para desenvolver uma interface de linha de comando simples para ocultação de dados em binários compilados para as plataformas x86 e AMD64, oferecendo suporte tanto para ELF quanto para PE. Além de operações básicas de injeção e extração de mensagens, o utilitário também oferece um comando para fazer uma análise de um binário e determinar quantos bits podem ser inseridos nele (WOODRUFF, 2020).

3 DESENVOLVIMENTO DA TÉCNICA

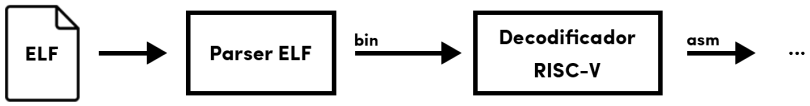
Durante esse capítulo serão detalhadas as técnicas e passos utilizados para atingir os objetivos citados no Capítulo 1, contando com o desenvolvimento de ferramentas de apoio, análise da viabilidade da aplicação de métodos esteganográficos existentes na ISA do RISC-V e a escolha da técnica a ser implementada por este trabalho.

3.1 FERRAMENTAL DE SUPORTE

A fim de alcançar os objetivos propostos por este trabalho foi necessário estudar tópicos correlacionados e desenvolver aplicações utilitárias quando não houvessem projetos `open source` capazes de suprir a necessidade. Estes módulos, além de contribuírem para a consecução dos objetivos deste estudo, possuem a versatilidade de serem empregados de forma autônoma em outras aplicações ou pesquisas correlatas, potencializando assim sua utilidade e alcance no campo da esteganografia digital.

Ao operar sobre arquivos binários compilados é necessário interpretar os dados brutos a fim de extrair a informação que será utilizada para determinar os critérios com os quais a técnica de esteganografia irá performar a inserção dos dados. Para isso, o arquivo ELF deve ser ter seus metadados analisados com o objetivo de encontrar as seções de instruções, em seguida as sequências de bits correspondentes às instruções devem ser separadas com seus respectivos tamanhos e por fim devem ser decodificadas para terem sua semântica revelada. Esse fluxo é detalhado na Figura 3 e será utilizado como base para o desenvolvimento das ferramentas de apoio.

Figura 3 – Fluxo de Interpretação de um Arquivo ELF.



Fonte: Elaborado pelo Autor.

Em posse da informação relacionada a cada instrução é possível criar a lógica referente à técnica esteganográfica que está sendo desenvolvida, operar sobre as instruções e retornar elas diretamente para o arquivo ELF seguindo o caminho reverso detalhado no fluxo. Os passos apresentados são genéricos e podem ser reaproveitados para qualquer método de esteganografia que vise modificar instruções em um arquivo ELF compilado para RISC-V, portanto o ferramental de suporte só precisa ser desenvolvido uma vez.

3.1.1 *Parser* para arquivos ELF

A partir da decisão de trabalhar diretamente em arquivos binários modificando sua estrutura e conteúdo base com o intuito de esconder informações se cria a necessidade de poder abrir, interpretar, analisar e editar arquivos no formato de escolha. Como definido no Capítulo 2, este trabalho foca em explorar o formato ELF utilizado amplamente em sistemas Unix. Apesar da existência de utilitários capazes de executar essa tarefa a decisão por desenvolver o *parser* dentro projeto foi tomada pela facilidade em analisar apenas as informações estritamente necessárias para encontrar trechos de instruções de máquina dentro do arquivo.

O funcionamento do *parser* é baseado na leitura de cabeçalhos contendo as informações sobre as seções e segmentos de um arquivo ELF, codificados de forma numérica correspondente a tabelas definidas na especificação do formato. Primeiramente fazemos a leitura do *ELF Header* para nos certificarmos de que o arquivo foi compilado para a arquitetura RISC-V e extraímos informações importantes para os próximos passos, como a ordenação dos *bytes*, o número de seções e segmentos e o índice da tabela de símbolos. Para os objetivos da aplicação desenvolvida, os segmentos serão ignoradas pelo algoritmo, dado que eles possuem dados dinâmicos utilizados pelo Sistema Operacional (SO) e a implementação requer a busca de endereços estáticos, presentes nas seções.

Em seguida fazemos a leitura sequencial dos cabeçalhos de seções, utilizando os dados sobre permissões e tipo para identificar trechos de código no arquivo. Ao encontrar uma seção de código é possível pular para seu endereço relativo e retornar os blocos de 32 *bits* correspondentes às instruções de máquina contidas no código. Esse passo do algoritmo é realizado até que a seção seja esgotada, ao ponto em que será buscada a próxima seção para que se repitam os passos anteriores até o esgotamento do próprio arquivo. Com esse processo finalizado temos a lista completa de instruções contidas no programa e já é possível manipulá-las.

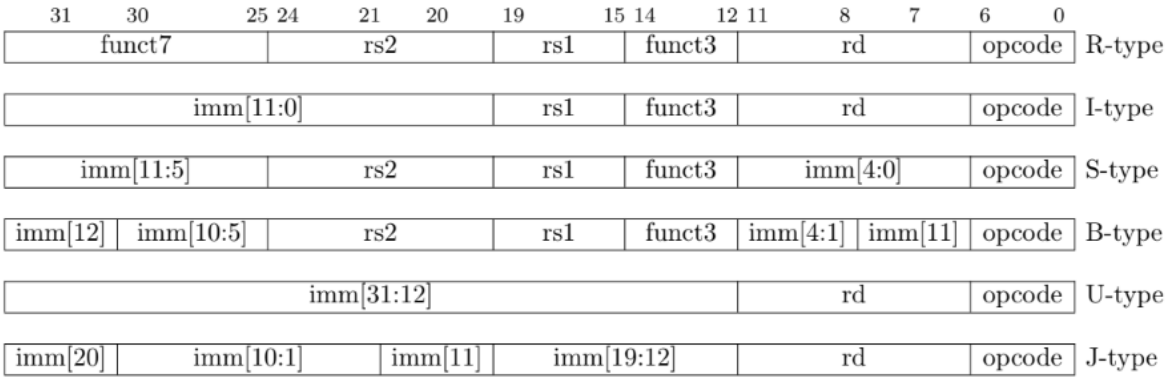
3.1.2 Decodificador e Codificador para instruções RISC-V

Após a extração completa de todas as instruções presentes em um binário resta uma lista de números com 32 *bits* sem um significado claro para um leitor humano. Para que qualquer técnica esteganográfica seja aplicada sobre uma mídia digital é necessário compreender a semântica dos dados, de modo a evitar o corrompimento acidental da informação contida. Com o propósito de decodificar as instruções para um formato mais legível por seres humanos foi desenvolvido um decodificador de instruções capaz de receber um objeto *bytes* e retornar a instrução correspondente escrita em Assembly.

A especificação do RISC-V introduz conjuntos de instruções base e extensões de propósitos específicos com suas próprias instruções que evoluem com o passar das versões da plataforma, obedecendo formatos de codificação de instruções que definem a

posição e semântica dos campos dentro dos 32 bits disponíveis. O conjunto base contém 6 formatos definidos capazes de codificar as instruções mais básicas para o funcionamento de um processador RISC-V, esses formatos estão descritos nas especificações técnicas da arquitetura (WATERMAN; ASANOVIĆ, 2019).

Figura 4 – Formatos de Instrução disponíveis na ISA base do RISC-V.



Fonte: Retirado de (WATERMAN; ASANOVIĆ, 2019).

Para os objetivos deste trabalho cobrir estes 6 tipos é suficiente, dado que formatos específicos de extensões são menos comuns em programas compilados e podem variar conforme os processadores que adotam a arquitetura. O processo de decodificação foi realizado de forma a identificar inicialmente a qual formato uma instrução pertence utilizando o campo *opcode* que compreende os 7 primeiros bits de cada conjunto de 32 bits gerados pelo compilador. Após a identificação do formato, os demais campos são separados utilizando operações binárias, respeitando particularidades de campos com sinal aritmético para valores numéricos e consultando as tabelas de operações descritas na especificação da ISA. Com todos os campos já traduzidos a instrução é montada em Assembly já com seu mnemônico, registradores e operandos identificados.

Após a decodificação é criado também um objeto editável contendo os campos de cada instrução para que seja possível alterar os valores representados em alto nível, expondo um método capaz de realizar a transformação reversa, utilizando os campos da instrução para gerar sua contraparte binária. Dessa forma, uma instrução pode ser decodificada, alterada e então recodificada de volta para seu formato binário, podendo ser sobrescrita no arquivo fonte. Esse processo em conjunto com o *parser* de arquivos ELF permite a livre manipulação de qualquer instrução presente em um binário compilado para RISC-V e é a base utilizada posteriormente pelo algoritmo esteganográfico para inserir dados na mídia de destino.

A implementação do decodificador e codificador foi baseada no trabalho de pesquisadores da Universidade de Columbia, desenvolvido na linguagem Javascript com propósitos educacionais (PORQUET-LUPINE; SAKAI; SOHAL, 2023). O projeto *rvcodec.js* visa faci-

litar o ensino da ISA disponibilizando uma ferramenta online para conversão de instruções e cobre as principais extensões da plataforma além do conjunto base.

3.2 APLICAÇÃO DE TÉCNICAS CONHECIDAS DE ESTEGANOGRAFIA

Nesta seção está descrito todo o processo, análise e aplicação de técnicas já conhecidas na litetura para codificação de informação em arquivos binários, a fim de verificar sua viabilidade quando transpostas para a arquitetura RISC-V. Foram analisadas apenas técnicas cuja aplicação não envolvem alteração do tamanho do arquivo final ou realocação de endereços ou etiquetas, de forma a manter a viabilidade da implementação para uma gama maior de casos de uso.

3.2.1 Alocação de Registradores

A especificação atual da arquitetura RISC-V prevê 32 registradores de 32 bits para serem utilizados com valores inteiros e outros 32 registradores disponíveis para valores de ponto flutuante, sendo que apenas 8 registradores em cada um desses conjuntos é reservado para alocação de argumentos. Apesar de teoricamente ser possível codificar bits utilizando a escolha dos registradores alocados para cada chamada de função, a convenção de chamadas prevê que esses registradores sejam escolhidos de forma ordenada e crescente, impedindo o uso prático dessa abordagem para implementações (RISC-V FOUNDATION, 2023).

3.2.2 Escalonamento de Instruções

Certas operações de alto nível quando executadas pelo processador passam por um desmembramento em duas ou mais instruções nativas que podem ser independentes entre si, gerando uma decisão de ordenação que pode ser explorada com o intuito de codificar bits de informação. Tal técnica foi utilizada por Anckaert *et al* (2004) com a arquitetura alvo sendo IA-32, uma arquitetura CISC que dispõe de um número elevado de instruções redundantes ou semelhantes que podem performar operações equivalentes, portanto seus resultados são satisfatórios.

Entretanto, quando o contexto é alterado para uma arquitetura RISC o número de permutações equivalentes a cada operação de alto nível tende a ser reduzido, impactando na eficiência desse método. Além disso o esforço necessário a ser empregado para essa técnica também é custoso pois requer a implementação de uma ferramenta capaz de gerar todas as permutações possíveis para cada procedimento a que se deseje explorar. A partir do entendimento de que esse método possui um alto custo temporal de implementação e a expectativa de resultados é baixa, o escalonamento de instruções não será abordado durante a execução desse trabalho.

3.2.3 Substituição de Instruções Equivalentes

A estratégia de utilizar escolhas dentro de uma única instrução como estrutura de codificação binária é amplamente utilizada por autores desde os primeiros trabalhos realizados na área, El-Khalil e Keromytis (2004) baseiam sua solução em criar tabelas de instruções equivalentes e atribuir um valor para cada alternativa, dessa forma para cada instrução pertencente a um conjunto de instruções equivalentes, podemos codificar $\log_2 n$ bits, sendo n o tamanho do conjunto. Os autores catalogaram 18 conjuntos de equivalência dentro da ISA `x86`, variando entre operações aritméticas e binárias, com equivalências produzidas por ordem de operandos e inversas matemáticas de instruções.

Devido as diferenças filosóficas entre arquiteturas RISC e CISC, a aplicação direta dessa técnica possui uma eficiência reduzida, com um número menor de conjuntos equivalentes a serem explorados por um algoritmo de substituição. Muitas das redundâncias comuns em outras ISAs foram removidas propositalmente durante o desenvolvimento do RISC-V, restando apenas aquelas que seriam impossíveis de remover como as equivalências lógicas e aritméticas axiomáticas. Dado isso, foram buscadas instruções com propriedades comutativas no conjunto base de instruções, resultando em cinco grupos contendo exatamente duas instruções cada que estão detalhadas na Tabela 2. Por termos apenas duas instruções por grupo, é possível codificar $\log_2 2$ bits por cada ocorrência no programa.

Nome do grupo	**Instrução**
add-group	add rd, rs1, rs2
	add rd, rs2, rs1
and-group	and, rd, rs1, rs2
	and, rd, rs2, rs1
or-group	or, rd, rs1, rs2
	or, rd, rs2, rs1
beq-group	beq, rd, rs1, rs2
	beq, rd, rs2, rs1
bne-group	bne, rd, rs1, rs2
	bne, rd, rs2, rs1

Tabela 2 – Conjuntos de instruções equivalentes no RISC-V

Fonte: Elaborado pelo autor (2023)

As instruções encontradas são operações comuns e presentes em todo programa de computador, o que indica uma possível abundância de espaço para codificação de informação. Ademais, não existe uma convenção da arquitetura que denote qual deva ser

a ordem dos operandos de uma determinada instrução, dessa forma podemos descrever o passo a passo para codificar bits utilizando essa estratégia utilizando o Algoritmo 1.

```
1  P ← {...};
2  I ← {...};
3  M ← "01000110010011000100000100000000";
4  message_index ← 0;
5  while message_index ≤ |M| do
6  |  encoded ← false;
7  |  while encoded = false do
8  |  |  instruction ← next(P);
9  |  |  if instruction not in I then
10 |  |  |  continue;
11 |  |  end
12 |  |  if instruction.rs1 = instruction.rs2 then
13 |  |  |  continue;
14 |  |  end
15 |  |  if char = 1 then
16 |  |  |  if not (instruction.rs1 > instruction.rs2) then
17 |  |  |  |  swap(instruction.rs1, instruction.rs2);
18 |  |  |  end
19 |  |  else
20 |  |  |  if not (instruction.rs1 < instruction.rs2) then
21 |  |  |  |  swap(instruction.rs1, instruction.rs2);
22 |  |  |  end
23 |  |  end
24 |  |  encoded ← true;
25 |  end
26 end
27 return P;
```

Algoritmo 1: Codificação de bits utilizando substituição de instruções equivalentes

Inicialmente define-se P como o conjunto contendo todas as instruções do programa alvo ordenadas pelo seu endereço relativo ao arquivo, I como o conjunto de instruções capazes de codificar um bit (vistos na Tabela 2) e M como a mensagem binária a ser inserida no arquivo. Foi definido que para esta implementação o critério de final da mensagem é dado por um caractere American Standard Code for Information Interchange (ASCII) nulo, representado por 8 bits de valor zero. Em seguida itera-se por cada caractere da mensagem buscando a próxima instrução que esteja contida em I. Ao encontrar uma instrução candidata, caso seus dois operandos sejam idênticos é necessário buscar outra pois é impossível codificar uma escolha nessa situação. Do contrário, é possível codificar um bit ordenando os operandos de acordo com um critério pré estabelecido, durante a execução deste trabalho usaremos como critério a função f definida por:

$$f(rs1, rs2) = \begin{cases} 1 & \text{se } rs1 > rs2 \\ 0 & \text{se } rs2 > rs1 \\ \phi & \text{caso contrário} \end{cases} \quad (1)$$

Ao final da iteração pelos caracteres da mensagem o conjunto P representa a lista de instruções já com a mensagem embutida em si, a partir desse ponto o codificador RISC-V pode aplicar o processo reverso para transformar as instruções em linguagem de montagem para código de máquina e escrever o arquivo final que representa o estego-objeto.

```
1  P ← {...};
2  I ← {...};
3  M ← string();
4  last_char ← string() ;
5  while last_char ≠ "00000000" do
6  |  if last_char.length = 8 then
7  |  |  last_char ← string() ;
8  |  end
9  |  instruction ← next(P);
10 |  if instruction not in I then
11 |  |  continue;
12 |  end
13 |  if instruction.rs1 = instruction.rs2 then
14 |  |  continue;
15 |  end
16 |  if instruction.rs1 > instruction.rs2 then
17 |  |  M += "1";
18 |  else
19 |  |  M += "0";
20 |  end
21 end
22 return M;
```

Algoritmo 2: Decodificação de bits utilizando substituição de instruções equivalentes

Para decodificar uma mensagem propõe-se um algoritmo semelhante ao anterior, tomando P como o conjunto de todas as instruções do programa, I como o conjunto de instruções codificáveis e M como uma *string* vazia que ao final da execução receberá a mensagem decodificada. Entretanto, após encontrar uma instrução candidata para a técnica um bit é extraído a partir da ordem de seus operandos e a concatenação final de todos os bits contém a mensagem, respeitando o critério de parada de caractere nulo

controlado por uma variável que é verificada a cada 8 bits para uma representação em ASCII.

Dado que as outras técnicas exploradas não produziram resultados por particularidades da arquitetura, a aplicação desenvolvida por esse trabalho teve como foco a utilização da substituição de instruções equivalentes como mecanismo de codificação de dados.

4 DESENVOLVIMENTO DA APLICAÇÃO

Na extensão deste capítulo serão abordadas as técnicas, decisões de projeto e desenvolvimento da aplicação proposta nos objetivos do trabalho, implementando o método de substituição de instruções equivalentes, descrito no Capítulo 3 como estrutura de codificação de dados e apresentando um *software* capaz de inserir uma mensagem arbitrária dentro de um arquivo binário de código compilado.

4.1 ESCOLHA DE LINGUAGEM E DEFINIÇÕES GERAIS DO PROJETO

4.1.1 Linguagem e Ambiente

Para todos os desenvolvimentos práticos foi definido o uso da linguagem Python, conhecida por sua simplicidade e clareza de código, o que facilita a compreensão para os pesquisadores e desenvolvedores de diversas áreas, em sua versão 3.10. Além disso, a vasta gama de bibliotecas e módulos disponíveis em Python, como `struct`, `pickle`, e `array`, oferece funcionalidades prontas para o tratamento de dados binários, economizando tempo e esforço no desenvolvimento. A portabilidade do Python também é uma vantagem, já que a linguagem é suportada em diferentes sistemas operacionais, garantindo que os resultados do projeto sejam facilmente reproduzíveis. Por fim, a comunidade Python ativa e os recursos de documentação tornam mais simples a resolução de problemas e o suporte durante o desenvolvimento do projeto.

Até o momento da publicação dessa monografia não existe uma forma obrigatória definida pela linguagem de como estruturar seus projetos, portanto foram seguidas as recomendações dadas no livro *The Hitchhiker's Guide to Python* que são utilizadas em larga escala pela comunidade (REITZ; SCHLUSSER, 2016) em conjunto com o gerenciador de dependências Poetry utilizado para distribuir de forma homogênea a aplicação em diversas máquinas (POETRY..., s.d.).

4.1.2 Estrutura de módulos

A aplicação está organizada em módulos Python com responsabilidades distintas a fim de hermetizar cada funcionalidade em um único local e evitar dependências desnecessárias e facilitar a manutenabilidade de cada módulo. Em seguida estão descritos os módulos com relevância para o entendimento da implementação da técnica proposta no capítulo passado.

4.1.2.1 `arch.riscv`

Esse módulo contém os arquivos responsáveis por todo tipo de transformação e operação relacionadas com a arquitetura RISC-V, como o decodificador de instruções, codificador de instruções e os próprios formatos de instruções e seus campos. Todas as

classes foram desenvolvidas seguindo uma interface definida em um módulo de definição denominado `arch.common` para facilitar uma implementação futura de outras arquiteturas.

4.1.2.2 `elf`

Módulo responsável por decodificar arquivos no formato ELF a partir de uma estrutura de dicionários aninhados e enumerações que seguem a especificação técnica da extensão, oferecendo suporte para o *parsing* de segmentos, seções e tabelas de símbolos, além de um sumário do cabeçalho principal do arquivo.

4.1.2.3 `handlers`

Contém as funções responsáveis por receber os argumentos da execução do programa e executar a lógica de cada operação de alto nível disponível para o usuário. Para cada comando disponível para o usuário, existe um arquivo de *handler* homônimo encarregado de implementar entrada, processamento e saída daquele comando, essa estrutura é sugerida pela biblioteca `click` e facilita a adição de novos comandos na aplicação.

4.1.2.4 `utils.binary_manipulation`

Esse módulo simples contém algumas operações comuns para serem aplicadas em dados binários, como inversão de ordenamento de bytes, complemento de 2 e extensões de sinal. Outros módulos como o `arch.riscv` e `elf` se aproveitam dessas funções disponíveis para simplificar suas próprias lógicas internas.

4.1.2.5 `utils.crypto`

Durante o processo de codificação e decodificação de mensagens embutidas em arquivos binários existe também uma camada de criptografia que é implementada por este módulo, que na prática apenas encapsula funcionalidades da biblioteca `pycryptodome` de forma mais amigável para um programador, oferecendo operações de alto nível compostas por uma série de operações menores a fim de prover ao usuário uma forma segura e simples de criptografar e descriptografar porções arbitrárias de dados.

4.1.3 Estrutura de Execução da Interface

A aplicação foi organizada de forma a expor suas funcionalidades por meio de uma interface de linha de comando, utilizando a biblioteca `click` que permite a declaração de comandos a partir anotações de decoradores em funções da linguagem. O comando lê a entrada fornecida pelo usuário verificando se todos os parâmetros obedecem obrigatoriedade e tipos definidos pela interface, levantando exceções legíveis caso algum dado passado falhe na checagem. Em seguida o *handler* inicia o processamento do comando e retorna a saída para o usuário, encerrando o fluxo de uma chamada do programa.

4.1.4 Comandos Disponíveis

4.1.4.1 `encode`

O comando `encode` é responsável por receber uma mensagem em texto plano, criptografá-la e inserir-la no arquivo de fonte, gerando ao final o estego-objeto. Para os propósitos deste trabalho foi utilizada criptografia simétrica com o algoritmo Advanced Encryption Standard (AES) sendo instanciado em *Output Feedback Mode*, uma modalidade do AES que trabalha de forma serial utilizando o vetor de inicialização para gerar o texto cifrado de forma iterativa, com uma operação *XOR* entre o texto e um bloco cifrado a partir da chave, gerando uma distribuição de probabilidade pseudoaleatória para cada bit da mensagem cifrada final (DWORKIN, 2001). Todas as chaves são derivadas a partir de uma senha escolhida pelo usuário utilizando Password-Based Key Derivation Function 2 (PBKDF2), uma função computacional capaz de derivar chaves pseudoaleatórias a partir de uma entrada em texto plano, utilizando como função pseudoaleatória Hash-based Message Authentication Code (HMAC) em sua forma SHA512, com 2^{15} iterações e um *salt* de 16 bytes (TURAN *et al.*, 2010).

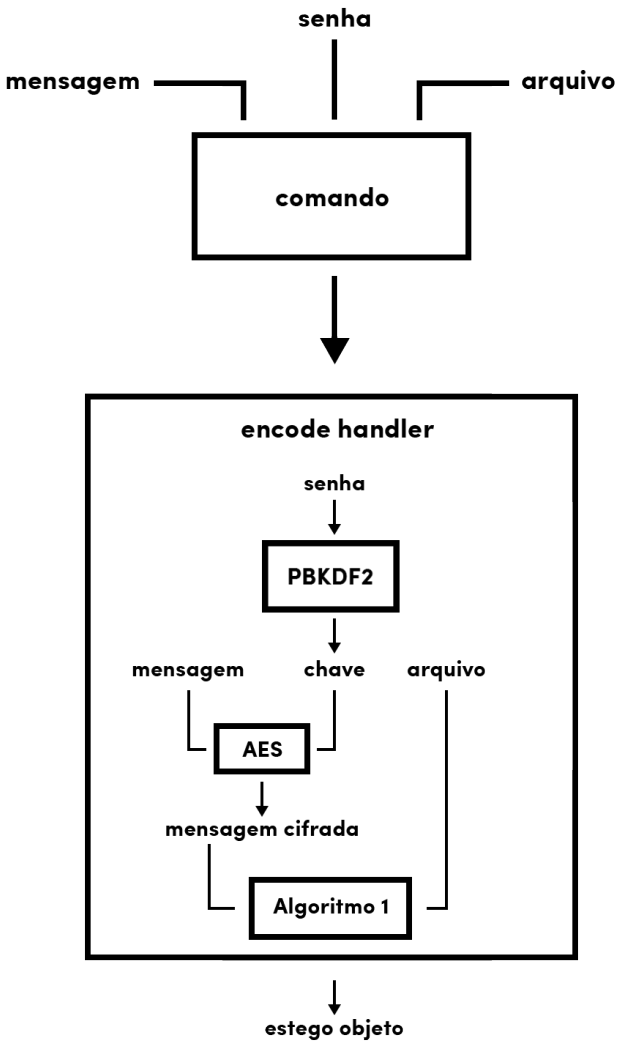
Após criptografada a mensagem é concatenada ao *salt*, codificada em *base64* e inserida conforme definido no Algoritmo 1, adicionando um caractere de final de mensagem para delimitar o tamanho total do texto inserido e utilizar como critério de parada no processo de decodificação. Nesse passo foi realizada uma otimização de implementação no algoritmo, o parâmetro de entrada P é fornecido a partir de uma função do tipo *generator*, que age como um objeto iterável retornando a cada iteração uma instrução do programa. Dessa forma, mesmo que seja utilizado um arquivo de tamanho elevado as instruções são trazidas para memória apenas quando serão utilizadas, resultando em diminuição drástica do uso de recursos para execuções em grandes arquivos.

O arquivo modificado é salvo em caminho alternativo ao original, a partir de uma cópia do original, alterando apenas as instruções codificadas no programa o que diminui o número de operações em disco realizadas pela implementação, dessa forma o usuário mantém o objeto de cobertura intacto e recebe um estego objeto recém criado. Caso a mensagem exceda a capacidade de codificação do arquivo informado é levantada uma exceção que alerta o usuário sobre o ocorrido e o arquivo não é alterado em nenhum aspecto. O funcionamento completo do comando de codificação está ilustrado na Figura 5.

4.1.4.2 `decode`

O processo de decodificação é realizado de forma inversa a codificação, recebendo um arquivo binário que se presume ser um estego objeto criado com a mesma técnica e uma senha para descriptografar a mensagem. Inicialmente buscamos toda a mensagem criptografada contida no arquivo, utilizando o Algoritmo 2 como base até encontrar um

Figura 5 – Esquemático do funcionamento do processo de codificação.



Fonte: Imagem produzida pelo Autor

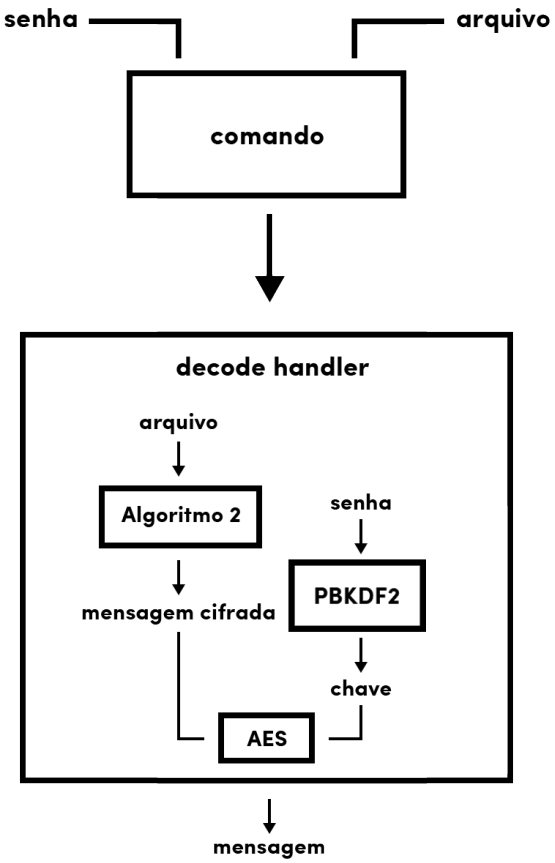
caractere de parada, caso nenhum caractere de parada seja encontrado a aplicação levanta uma exceção alertando o usuário de que o arquivo não contém dados relevantes inseridos em si. Ao encontrar uma mensagem geramos uma chave a partir da senha utilizando PBKDF2 e o *salt* contido no início da mensagem, ao ponto em que podemos descriptografar a mensagem utilizando o AES com a chave derivada e retornar a mensagem em texto plano para o usuário. O funcionamento completo do comando de codificação está ilustrado na Figura 6.

4.1.5 Execução e Utilização da Aplicação

Para garantir a transparência e facilitar o acesso e reprodutibilidade do trabalho desenvolvido, o código completo foi disponibilizado publicamente em um repositório *Git*[1].

[1] Repositório *Git* com o código fonte: https://github.com/leviosar/tcc

Figura 6 – Esquemático do funcionamento do processo de decodificação.



Fonte: Imagem produzida pelo Autor

A utilização efetiva da aplicação requer que o usuário tenha instalado o Python na versão 3.11 ou superior, bem como o gerenciador de pacotes Poetry. As etapas para instalação e execução são detalhadas e podem ser facilmente seguidas por meio das instruções no arquivo `README.md` encontrado no diretório *src* do projeto. Este repositório não apenas permite que interessados reproduzam, explorem e contribuam para a aplicação, mas também serve como um recurso prático para aqueles que buscam aplicar a técnica desenvolvida em seus próprios estudos.

Como mencionado anteriormente a aplicação implementada utiliza uma interface de linha de comando para responder à entrada do usuário, seguindo os padrões estabelecidos pela especificação Portable Operating System Interface (POSIX) para interpretação de opções, argumentos e caminhos de sistema, no trecho de código abaixo está descrita a utilização do programa a partir de sua interface.

```
1 > python -m steganossaurus --help
2
3 Usage: python -m steganossaurus [OPTIONS] COMMAND [ARGS]...
```

```
4
5 Options:
6   --help  Show this message and exit.
7
8 Commands:
9   decode   Recover a message from the given FILE, decrypting it with a
     required password.
10  encode   Embeds a message into the given FILE, encrypting it with a
     required password.
```

Primeiramente pode ser utilizado o argumento `help` sem nenhum comando para retornar mais informações sobre o programa, resultando em um curto guia de uso dos argumentos e disponibilização da lista de comandos.

```
1 > python -m steganossaurus encode --help
2
3 Usage: python -m steganossaurus encode [OPTIONS] FILE MESSAGE
4
5   Embeds a message into the given FILE, encrypting it with a required
     password.
6
7 Options:
8   -o, --output PATH           Path to output file, created if not exists.
9   --log-level                 Level of data to de logged.
10  --password                  Password used to encrypt the message.
11  --help                      Show this message and exit.
12
13 > python -m steganossaurus encode ./input "steganographia" -o ./output
14
15 Password: **********
16 Repeat for confirmation: **********
17 Message was encoded at ./output
```

Em seguida o usuário pode chamar do comando `encode` com o argumento `help` que retorna instruções específicas para utilização do comando, como as opções disponíveis, a ordem e o nome dos argumentos. O comando `encode` então é invocado utilizando como argumentos um arquivo de entrada, uma mensagem para ser codificada pelo algoritmo e como opção um arquivo de saída. A aplicação então requer a digitação e confirmação de uma senha para o a mensagem criptografada e processa a solicitação do usuário, inserindo a mensagem no arquivo utilizando o Algoritmo 1 e gerando um estego-objeto no caminho fornecido.

```
> python -m steganossaurus decode --help

Usage: python -m steganossaurus decode [OPTIONS] FILE

  Recover a message from the given FILE, decrypting it with a required
   password.

Options:
  --log-level               Level of data to de logged.
  --password TEXT           Password used to encrypt the message.
  --help                    Show this message and exit.

> python -m steganossaurus decode .\output

Password: **********
Repeat for confirmation: **********
Message found: steganographia
```

Para o processo de decodificação também é oferecida a opção de consultar a documentação do comando a partir da opção `help` que retorna todas as informações necessárias para a utilização do comando `decode`. Para finalizar o ciclo de vida da aplicação o usuário invoca o comando de decodificação passando como argumento o estego-objeto gerado no passo anterior, a aplicação solicita a senha e caso confirmada extrai a mensagem utilizando o Algoritmo 2, descriptografa com a chave gerada a partir da senha informada e retorna a mensagem em texto plano para o usuário.

5 DISCUSSÃO DE RESULTADOS

5.1 APLICAÇÕES PARA EXPERIMENTOS

Para avaliar os resultados obtidos pelas técnicas de esteganografia e validar o ferramental de apoio desenvolvido durante este trabalho foram utilizado dois conjuntos distintos de programas como objetos de experimento. O primeiro conjunto contém a compilação de 2 sistemas operacionais, o segundo conjunto é composto 22 programas utilitários de bancos de dados. Os sistemas operacionais foram selecionados devido a alta gama de operações distintas realizadas por esse tipo de programa durante sua execução. Os utilitários com enfoque para bancos de dados foram escolhidos por possuirem em seu código algoritmos variados de ordenação, busca e transformação de dados, com o intuito de entender se um programa altamente focado em operações em disco é mais suscetível a técnicas de esteganografia.

Para ambos os conjuntos de programas foi utilizado o processo de compilação cruzada a partir de uma máquina x86 tendo como arquitetura alvo o RISC-V para eliminar a necessidade de possuir uma placa física implementando RISC-V. No primeiro conjunto os softwares utilizados foram o Embedded Parallel Operating System (EPOS), desenvolvido com o propósito de ser um SO enxuto e eficiente capaz de ser executado em microprocessadores de sistemas embarcados e o Linux, um SO de propósito geral utilizado na maioria dos dispositivos pelo mundo e que conta com funcionalidades mais variadas e robustas do que o primeiro. Para o segundo conjunto foram utilizados os utilitários gerados na compilação do Sistema Gerenciador de Banco de Dados (SGBD) PostgreSQL, que incluem o próprio SGBD e outros módulos utilizados para funcionalidades específicas do software. As Tabelas 3 e 4 contém os nomes, versões e tamanhos de arquivos (descritos em bits) de todos os softwares utilizados para os experimentos, separados em seus conjuntos.

Programa	**Versão**	**Tamanho**
Linux	v6.5	150792320
EPOS	v2.2.2	288128

Tabela 3 – Conjunto 1 de programas utilizados nos experimentos

Fonte: Elaborado pelo autor (2023)

Programa	**Versão**	**Tamanho**
initdb	v16.0.0	1252608
pgbench	v16.0.0	1502784
pg_amcheck	v16.0.0	809536
pg_archivecleanup	v16.0.0	372736
pg_basebackup	v16.0.0	1197632
pg_checksums	v16.0.0	657152
pg_config	v16.0.0	360512
pg_controldata	v16.0.0	475456
pg_ctl	v16.0.0	566784
pg_dump	v16.0.0	2857344
pg_dumpall	v16.0.0	1003264
pg_receivewal	v16.0.0	876736
pg_recvlogical	v16.0.0	841408
pg_resetwal	v16.0.0	524928
pg_restore	v16.0.0	1414208
pg_rewind	v16.0.0	1252928
pg_test_fsync	v16.0.0	405504
pg_test_timing	v16.0.0	331136
pg_upgrade	v16.0.0	1247936
pg_verifybackup	v16.0.0	864768
pg_waldump	v16.0.0	843392
psql	v16.0.0	3672768

Tabela 4 – Conjunto 2 de programas utilizados nos experimentos

Fonte: Elaborado pelo autor (2023)

5.2 EXPERIMENTOS DE VALIDAÇÃO DO DECODIFICADOR PARA RISC-V

Os dois conjuntos de testes foram utilizados como método de avaliação da capacidade de decodificação do módulo apresentado, utilizando como métrica a razão entre a quantidade de instruções presentes no programa e a quantidade de instruções que foram decodificadas após o processo. O esperado desse experimento era verificar se os 6 formatos de instrução da ISA base cobrem uma porção suficientemente grande de um binário arbitrário para que seja possível obter taxas de codificação satisfatórias posteriormente.

Analisando resultados do experimento presentes nas Tabelas 6 e 5 pode-se notar que a cobertura da ISA base ocupa cima de 60% das seções de código do programa, o que indica o potencial para níveis sólidos de codificação de informação apenas com instruções desses formatos.

Programa	**Percentual de Instruções Decodificadas**
initdb	76.25%
pgbench	66.07%
pg_amcheck	61.71%
pg_archivecleanup	64.58%
pg_basebackup	68.58%
pg_checksums	63.47%
pg_config	67.76%
pg_controldata	65.86%
pg_ctl	68.65%
pg_dump	68.77%
pg_dumpall	75.62%
pg_receivewal	63.36%
pg_recvlogical	64.32%
pg_resetwal	65.45%
pg_restore	68.59%
pg_rewind	66.74%
pg_test_fsync	61.10%
pg_test_timing	65.93%
pg_upgrade	76.21%
pg_verifybackup	61.90%
pg_waldump	63.22%
psql	69.99%

Tabela 5 – Resultados do Experimento do Decodificador para o Conjunto 2

Fonte: Elaborado pelo autor (2023)

Programa	Percentual de Instruções Decodificadas
Linux	63.55%
EPOS	70.63%

Tabela 6 – Resultados do Experimento do Decodificador para o Conjunto 1

Fonte: Elaborado pelo autor (2023)

5.3 EXPERIMENTOS DE VALIDAÇÃO DA TÉCNICA DESENVOLVIDA

5.3.1 Validação Quantitativa

Após a elaboração do algoritmo apresentado para codificação utilizando substituição de instruções equivalentes foi realizado um experimento com a motivação de examinar a quantidade de bits que poderiam ser codificados dentro de cada arquivo presente no conjunto de testes. Para isso uma pequena modificação foi realizada no algoritmo, incrementando um contador sempre que encontrada uma instrução explorável pela técnica de modo a obter no final o total de bits codificáveis. Observando os resultados expostos nas Tabela 7 e 8 é possível verificar que a técnica possui uma capacidade de codificação baixa quando comparada com os resultados obtidos para a arquitetura x86 por outros autores. A taxa de codificação média obtida por essa implementação foi de 0,03%, inferior aos 0,9% demonstrados por El-Khalil e Keromytis (2004) e os 3,7% apresentados por Anckaert *et al* (2004). Os resultados inferiores para a arquitetura alvo desse trabalho são explicados principalmente pelo tamanho dos conjuntos de substituições disponíveis, enquanto a arquitetura da Intel conta com 18 conjuntos contendo entre 2 a 5 instruções, o RISC-V apresenta apenas 5 conjuntos de 2 instruções cada, reduzindo tanto a frequência de aparição de instruções exploráveis no programa quanto a quantidade de bits codificados por instrução.

Programa	Bits Codificáveis	Porcentagem Codificável
Linux	69854	0.05%
EPOS	87	0.03%

Tabela 7 – Resultados do Experimento Para o Conjunto 1

Fonte: Elaborado pelo autor (2023)

Entretanto, apesar de não apresentar resultados satisfatórios para codificação de grandes quantidades de dados o experimento comprova a viabilidade do uso dessa técnica para inserção de uma quantidade reduzida de bits, possibilitando o uso do método para aplicações específicas que não tenham como requisito o uso de mensagens extensas.

Programa	**Bits Codificáveis**	**Porcentagem Codificável**
initdb	296	0.02%
pgbench	421	0.03%
pg_amcheck	180	0.02%
pg_archivecleanup	78	0.02%
pg_basebackup	448	0.04%
pg_checksums	111	0.02%
pg_config	64	0.02%
pg_controldata	58	0.01%
pg_ctl	131	0.02%
pg_dump	851	0.03%
pg_dumpall	327	0.03%
pg_receivewal	233	0.03%
pg_recvlogical	237	0.03%
pg_resetwal	131	0.02%
pg_restore	638	0.05%
pg_rewind	336	0.03%
pg_test_fsync	79	0.02%
pg_test_timing	88	0.03%
pg_upgrade	228	0.02%
pg_verifybackup	354	0.04%
pg_waldump	379	0.04%
psql	1493	0.04%

Tabela 8 – Resultados do Experimento Para o Conjunto 2

Fonte: Elaborado pelo autor (2023)

5.3.2 Validação Qualitativa

Retornando ao problema dos prisioneiros que retrata um modelo de comunicação invisível, já descrito no Capítulo 2, após termos apresentado tanto a técnica quanto sua implementação nos capítulos anteriores é possível realizar uma avaliação qualitativa dos resultados obtidos. Primeiramente podemos identificar que a implementação corresponde as prerrogativas do enunciado do problema, considerando que em uma comunicação a partir da metodologia proposta é utilizada uma chave privada definida previamente entre

as duas partes, um objeto de cobertura arbitrário e se gera um estego objeto que pode ser trafegado em canal público. Devido à propriedade pseudoaleatória tanto dos bits gerados pela mensagem quanto operandos naturalmente codificados por instruções compiladas, é seguro afirmar que o carcereiro não teria a capacidade de identificar um subtexto malicioso dentro do estego objeto, e como assumimos a priori que a vigilância seria passiva, a destruição da integridade da mensagem não é preocupação deste estudo.

Existem ainda ataques comuns a classe de técnica utilizada na implementação deste trabalho descritos na litetura, que buscam encontrar assinaturas ou características comuns à transformações de código aplicadas (ANCKAERT *et al.*, 2004). Um atacante buscando detectar essas assinaturas pode buscar pela presença de instruções pouco utilizadas em uma arquitetura a fim de identificar um padrão de substituição. Entretanto como o Algoritmo 1 não substitui instruções, trabalhando apenas com substituição de operandos, não há risco de ter essa propriedade explorada. Outra possibilidade é a detecção de uma frequência anormal de um certo tipo de instruções, também descartada pelo motivo anterior. Por fim, o atacante pode inspecionar o comportamento dos *jumps* entre endereços do programa, buscando valores discrepantes que não seria naturalmente endereçados por compiladores. Tal vulnerabilidade também é inefetiva contra a aplicação desenvolvida pela ausência da alteração de valores imediatos em instruções de controle.

Nota-se então que os ataques já observados na bibliografia disponível não tem a capacidade de detectar as transformações realizadas pelo algoritmo proposto, o que não significa dizer que o mesmo seja indetectável, considerando que os ataques foram planejados para expor vulnerabilidades em técnicas aplicadas na arquitetura *x86* que possui uma gama de possibilidades de alteração superior. Em fato, a eficiência de codificação de instruções da arquitetura RISC-V simultâneamente favorece a avaliação qualitativa de transformações binárias enquanto desfavorece uma avaliação quantitativa, reduzindo o número de permutações equivalentes dentro do programa e tornando a codificação de informações mais silenciosa porém menos eficiente.

6 CONCLUSÃO

O campo da esteganografia evolui pela história humana antes mesmo do cunho de seu próprio termo, entretanto suas aplicações com utilização voltada para formatos de código compilado ainda não possuem técnicas tão difundidas em comparação com outras categorias de objetos de cobertura, tornando a própria execução deste trabalho um desafio considerável. Após revisar e analisar estudos formais na área foi possível utilizar os conhecimentos e técnicas obtidas para aplicar uma metodologia de codificação de dados já proposta em uma arquitetura ainda não explorada com sucesso.

O algoritmo proposto é capaz de explorar redundâncias provenientes de operações comutativas na arquitetura RISC-V para codificar um bit a cada operação suportada e sua implementação foi desenvolvida de forma a minimizar o uso de recursos para possibilitar a execução eficiente mesmo em arquivos de grande porte. Apesar do resultado positivo na viabilidade da aplicação é necessário ressaltar que sua eficácia é reduzida em comparação com as técnicas semelhantes apresentadas por El-Khalil e Keromytis (2004) e por Anckaert *et al* (2004), ambas implementadas em arquiteturas CISC, que favorecem as técnicas por possuírem uma codificação de instruções com redundância elevada.

Ao avaliar a técnica por outro prisma, relevando a baixa taxa de codificação e focando primariamente na detectabilidade e corretude quanto ao problema dos prisioneiros, entende-se que os pontos negativos podem ser mitigados com o uso do algoritmo em situações específicas, aproveitando-se da dificuldade em revelar a mensagem sem conhecimento da chave para inserir mensagens de tamanho reduzido em arquivos. Tais situações podem ser exploradas para embutir assinaturas e identificadores em *softwares* distribuídos comercialmente com o objetivo de rastrear violações de direitos autorais no combate à pirataria.

6.1 TRABALHOS FUTUROS

A execução deste trabalho revela algumas portas em aberto para a evolução das técnicas de esteganografia em código compilado, principalmente considerando a arquitetura RISC-V. Durante a revisão bibliográfica foram encontrados pontos de exploração com potencial para codificação de dados, que poderiam ser unidos a técnica desenvolvida para incrementar a taxa de codificação. Inicialmente existe a possibilidade de explorar as instruções `HINT`, reservadas no espaço de endereçamento da arquitetura como livres para uso de implementações, normalmente utilizadas para otimizações de performance mas que poderiam ser subvertidas para carregar bits de mensagens. Além disso, no decorrer da pesquisa realizada o foco foi aplicar transformações diretamente no conjunto de instruções base da arquitetura, deixando de fora diversas extensões de propósitos específicos que apesar de possuírem frequência menor de aparecimento em um programa arbirtrário podem aumentar a taxa total de codificação de dados.

Ainda no escopo da arquitetura RISC-V e da implementação atual, levando em consideração que a validação de detectabilidade da técnica foi realizada a partir de ataques comuns direcionados a binários compilados para *x86*, é necessário expor a técnica à uma gama maior de ataques que possam expor vulnerabilidades específicas da arquitetura para reforçar a confiabilidade do algoritmo.

Por fim, existem outras arquiteturas RISC amplamente utilizadas em dispositivos ao redor do mundo, principalmente da família ARM nas quais os conhecimentos adquiridos podem ser aplicados, resultando na portabilidade da técnica para um leque maior de usuários e possivelmente em resultados superiores se considerarmos que o endereçamento do RISC-V possui uma maior rigidez aos princípios RISC.

REFERÊNCIAS

AHO, Alfred V. *et al.* **Compilers: Principles, Techniques, and Tools (2nd Edition)**. [*S.l.*]: Addison Wesley, ago. 2006. Hardcover. ISBN 0321486811. Disponível em: <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%5C&path=ASIN/0321486811>.

ANCKAERT, Bertrand *et al.* Steganography for Executables and Code Transformation Signatures. *In:* 3506. 17TH International Conference on Information Security and Cryptology ICISC 2004. [*S.l.*: *s.n.*], dez. 2004. (LNCS, 3506), p. 425–439.

ANDERSON, Ross. Stretching the limits of steganography. *In:*_____. **Information Hiding**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. P. 39–48.

ASANOVIĆ, Krste; PATTERSON, David A. **Instruction Sets Should Be Free: The Case For RISC-V**. [*S.l.*], ago. 2014. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html>.

BLYTHE, Paul; FRIDRICH, Jessica. Secure Digital Camera. **Digital Forensic Research Workshop**, jan. 2009.

DAVID A. PATTERSON, John L. Henessy. **Computer Organization and Design: The Hardware/Software Interface**. 5th. [*S.l.*]: Elsevier, 2014.

DWORKIN, M J. **Recommendation for block cipher modes of operation :** [*S.l.*], 2001. DOI: <10.6028/nist.sp.800-38a>. Disponível em: <https://doi.org/10.6028/nist.sp.800-38a>.

EMBAR, Maya; MCHUGH, Louis F.; WESSELMAN, William R. Printer Watermark Obfuscation. *In:* PROCEEDINGS of the 3rd Annual Conference on Research in Information Technology. Atlanta, Georgia, USA: Association for Computing Machinery, 2014. (RIIT '14), p. 15–20. DOI: <10.1145/2656434.2656437>. Disponível em: <https://doi.org/10.1145/2656434.2656437>.

ERNST, Thomas. The Numerical-Astrological Ciphers in the Third Book of Trithemius's Steganographia. **Cryptologia**, Taylor & Francis, Inc., USA, v. 22, n. 4, p. 318–341, out. 1998. ISSN 0161-1194. DOI: <10.1080/0161-119891886957>. Disponível em: <https://doi.org/10.1080/0161-119891886957>.

EVSUTIN, Oleg; MELMAN, Anna; MESHCHERYAKOV, Roman. Digital Steganography and Watermarking for Digital Images: A Review of Current Research Directions. **IEEE Access**, v. 8, p. 166589–166611, 2020. DOI: <10.1109/ACCESS.2020.3022779>.

JOHNSON, Neil F.; JAJODIA, Sushil. Exploring steganography: Seeing the unseen. **Computer**, v. 31, n. 2, p. 26–34, 1998. DOI: <10.1109/MC.1998.4655281>.

JUDGE, James. Steganography: Past, Present, Future. **White Paper**, SANS Institute, 2001.

KATZENBEISSER, Stefan. **Information Hiding Techniques for Steganography and Digital Watermarking**. Edição: Fabien A. P. Petitcolas. [*S.l.*]: Artech House Print on Demand, 1999.

KERCKHOFFS, A. La cryptographie militaire. **Journal des Sciences Militaires**, p. 161–191, 1883.

EL-KHALIL, Rakan; KEROMYTIS, Angelos D. Hydan: Hiding Information in Program Binaries. *In:*_____. **Information and Communications Security**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. P. 187–199.

LINUX STANDARD BASE Chapter 4: Object Files. [*S.l.*: *s.n.*]. <https://refspecs.linuxbase.org/elf/gabi4+/ch4.intro.html>. Acessado em 17/06/2023.

LU, Kangjie; XIONG, Siyang; GAO, Debin. RopSteg: Program Steganography with Return Oriented Programming. *In:* (CODASPY '14), p. 265–272. DOI: <10.1145/2557547.2557572>. Disponível em: <https://doi.org/10.1145/2557547.2557572>.

MAHONEY, William *et al.* Leave It to Weaver. *In:* PROCEEDINGS of the 8th Software Security, Protection, and Reverse Engineering Workshop. San Juan, PR, USA: Association for Computing Machinery, 2018. (SSPREW-8). DOI: <10.1145/3289239.3291459>. Disponível em: <https://doi.org/10.1145/3289239.3291459>.

MICROSOFT PE Format. [*S.l.*: *s.n.*]. <https://learn.microsoft.com/pt-br/windows/win32/debug/pe-format>. Acessado em 17/06/2023.

PATTERSON, David A.; HENNESSY, John L. **Computer Organization and Design RISC-V Edition: The Hardware Software Interface**. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128122757.

PATTERSON, David A.; SEQUIN, Carlo H. RISC I: A Reduced Instruction Set VLSI Computer. *In:* PROCEEDINGS of the 8th Annual Symposium on Computer Architecture. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981. (ISCA '81), p. 443–457.

POETRY Documentation. [*S.l.*: *s.n.*]. <https://python-poetry.org/docs/>. Accessed: 2023-10-23.

PORQUET-LUPINE, Joël; SAKAI, Hikari N.; SOHAL, Abhi. Rvcodec.Js: An Educational Converter for RISC-V Instructions. *In:* PROCEEDINGS of the 54th ACM

Technical Symposium on Computer Science Education V. 2. Toronto ON, Canada: Association for Computing Machinery, 2023. (SIGCSE 2023), p. 1396. DOI: <10.1145/3545947.3576340>. Disponível em: <https://doi.org/10.1145/3545947.3576340>.

REEDS, Jim. SOLVED: THE CIPHERS IN BOOK III OF TRITHEMIUS'S STEGANOGRAPHIA. **Cryptologia**, Taylor Francis, v. 22, n. 4, p. 291–317, 1998. DOI: <10.1080/0161-119891886948>. eprint: <https://doi.org/10.1080/0161-119891886948>. Disponível em: <https://doi.org/10.1080/0161-119891886948>.

REITZ, K.; SCHLUSSER, T. **The Hitchhiker's Guide to Python: Best Practices for Development**. [*S.l.*]: O'Reilly Media, Incorporated, 2016. ISBN 9781491933176. Disponível em: <https://books.google.com.br/books?id=2ZggjwEACAAJ>.

RISC-V FOUNDATION. **RISC-V ABIs Specification**. [*S.l.*], out. 2023. Disponível em: <https://github.com/riscv-non-isa/riscv-elf-psabi-doc/releases/tag/draft-20231023-d1743f6adf4388dd3b6afe9708b5bb4473d88b4e>.

SIMMONS, Gustavus J. The Prisoners' Problem and the Subliminal Channel. *In:* **Advances in Cryptology: Proceedings of Crypto 83**. Edição: David Chaum. Boston, MA: Springer US, 1984. P. 51–67. ISBN 978-1-4684-4730-9. DOI: <10.1007/978-1-4684-4730-9_5>. Disponível em: <https://doi.org/10.1007/978-1-4684-4730-9_5>.

TOOL INTERFACE STANDARD (TIS) Executable and Linking Format (ELF) Specification. [*S.l.*: *s.n.*], Maio 1995. <https://refspecs.linuxbase.org/elf/elf.pdf>. Acessado em 17/06/2023.

TURAN, M S *et al.* **Recommendation for password-based key derivation :** [*S.l.*], 2010. DOI: <10.6028/nist.sp.800-132>. Disponível em: <https://doi.org/10.6028/nist.sp.800-132>.

WATERMAN, Andrew; ASANOVIĆ, Krste. **The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, Document Version 20191213**. [*S.l.*], dez. 2019. Disponível em: <https://github.com/riscv/riscv-isa-manual/releases/tag/Ratified-IMAFDQC>.

WATERMAN, Andrew; LEE, Yunsup *et al.* **The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0**. [*S.l.*], mai. 2014. Disponível em: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html>.

WOODRUFF, William. **steg86**. [*S.l.*]: GitHub, 2020. <https://github.com/woodruffw/steg86>.

WRIGHT, Craig. Detecting Hydan: Statistical Methods For Classifying The Use Of Hydan Based Stegonagraphy In Executable Files. *In:*

ZAIDAN, A.; BAHAA, Bilal; OTHMAN, Fazidah. New Technique of Hidden Data in PE-File with in Unused Area One. **International Journal of Computer and Electrical Engineering**, jan. 2009. DOI: <10.7763/IJCEE.2009.V1.100>.

ANEXO A – CÓDIGO FONTE

O código fonte desenvolvido para este projeto encontra-se integralmente disponível no repositório do autor no GitHub, acessível através do seguinte link: <https://github.com/leviosar/tcc>. O material abrange instruções detalhadas sobre a execução tanto dos experimentos quanto da aplicação.

ANEXO B – ARTIGO

Neste apêndice será apresentado o artigo no formato SBC, referente ao presente projeto.

Desenvolvimento de uma técnica de esteganografia explorando arquivos binários de código compilados para RISC-V.

João Vitor Maia Neves Cordeiro[1], **Jean Everson Martina**[1]

[1] Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

maia.tostring@gmail.com, jean.martina@ufsc.br

Abstract. *Stegranography is the craft of embedding information inside another source of data, hiding it from an external observer. Despite the extensive research on techniques built for image, audio and text steganography, when it comes to using executable files as cover objects the academic productions and tools are scarce. This article presents a method capable of embedding information inside an executable file compiled for the RISC-V architecture, without altering the semantics of the source program, and implements it as a command line application that may be used to a wide range of purposes, like digital signature, copyright protection and hidden communication. The results suggests that RISC architectures produces machine code capable of hiding a reduced amount of information when compared to CISC architecture, narrowing the scope of steganographic applications for those that do not require a high encoding rate.*

Resumo. *A esteganografia trabalha embutindo informação dentro de outra e escondendo essa transformação ao olhar de um observador ingênuo, de forma que não seja possível distinguir a mídia original do resultado transformado. Apesar do campo já possuir técnicas consolidadas para lidar com imagens, áudios e outros tipos de mídia, quando trata-se de arquivos de código binário a literatura e o ferramental atual são escassos. Dado esse contexto, esse trabalho apresenta o desenvolvimento de um algoritmo capaz de inserir informações dentro de um arquivo executável compilado para a arquitetura RISC-V sem alterar a semântica do programa fonte, bem como sua implementação em uma aplicação com interface em linha de comando, podendo ser utilizado para finalidades como assinatura digital de software, proteção de direitos autorais e comunicação oculta. Os resultados obtidos indicam que arquiteturas RISC produzem código de máquina capaz de abrigar uma quantidade reduzida de informação oculta quando comparadas com arquiteturas CISC, estreitando o alcance de aplicações de esteganografia para aquelas que não necessitam transmitir uma quantidade de dados elevada.*

1. Introdução

Esteganografia é a nomenclatura utilizada para um conjunto de técnicas, inicialmente desenvolvidas para espionagem, que atuam escondendo uma informação dentro de uma outra mídia, de modo que a mídia criada após a manipulação seja visualmente indistinguível da original para um observador externo.

A crescente disseminação de computadores a partir do século 20 recuperou diversas técnicas analógicas de espionagem que predatam a área da computação, adicionando o poder de processamento digital para aprimorar a efetividade e as aplicações

de tais técnicas. Notoriamente temos a criptografia como a principal expoente dessa transformação, estando presente na maior parte das aplicações modernas. Apesar de ser o maior caso a criptografia não foi a única área a passar por isso, as técnicas de esteganografia também foram afetadas pela introdução do fator digital [Johnson and Jajodia 1998].

Representações digitais de mídias diversas permitem que a informação seja processada e escondida de formas que uma mídia física, por sua natureza, não conseguiria suportar, fornecendo uma gama maior de técnicas e aplicações esteganográficas. Além disso, o meio digital permite a combinação das técnicas mais modernas de criptografia para esconder uma mensagem cifrada.

Apesar do comum ser utilizar técnicas esteganográficas em imagens, áudios e outros formatos de mídia de uso geral, os mesmos conceitos podem ser aplicados para qualquer formato de arquivo que contenha informação, ajustando ou criando novos algoritmos. Uma aplicação menos explorada atualmente é a esteganografia em arquivos de código binário, que pode ser utilizada para transmissão de mensagens secretas, marcas d'água digitais em software, verificação de integridade e autoria a partir de uma assinatura e até mesmo inclusão de trechos de código alternativo, maliciosos ou não, dentro do programa principal [Mahoney et al. 2018].

Trabalhos passados já obtiveram resultados no desenvolvimento e validação de técnicas esteganográficas em arquivos binários, notóriamente o projeto Hydan foi um dos precursores ao criar uma técnica para a arquitetura *x86* capaz de incorporar bits de informação em um arquivo na proporção de $\frac{1}{110}$. Em comparação, técnicas que se utilizam de mídias mais suscetíveis como imagens JPEG podem atingir uma eficiência de $\frac{1}{17}$, o que contribui para a aplicação de técnicas de obfuscação dentro do espaço de busca [El-Khalil and Keromytis 2004].

Esse artigo detalha o desenvolvimento de um algoritmo para a utilização de esteganografia em arquivos binários de código fonte em arquiteturas alternativas à x86, visando ampliar a aplicabilidade de técnicas esteganográficas em diferentes plataformas.

2. Conceituação e Revisão Bibliográfica

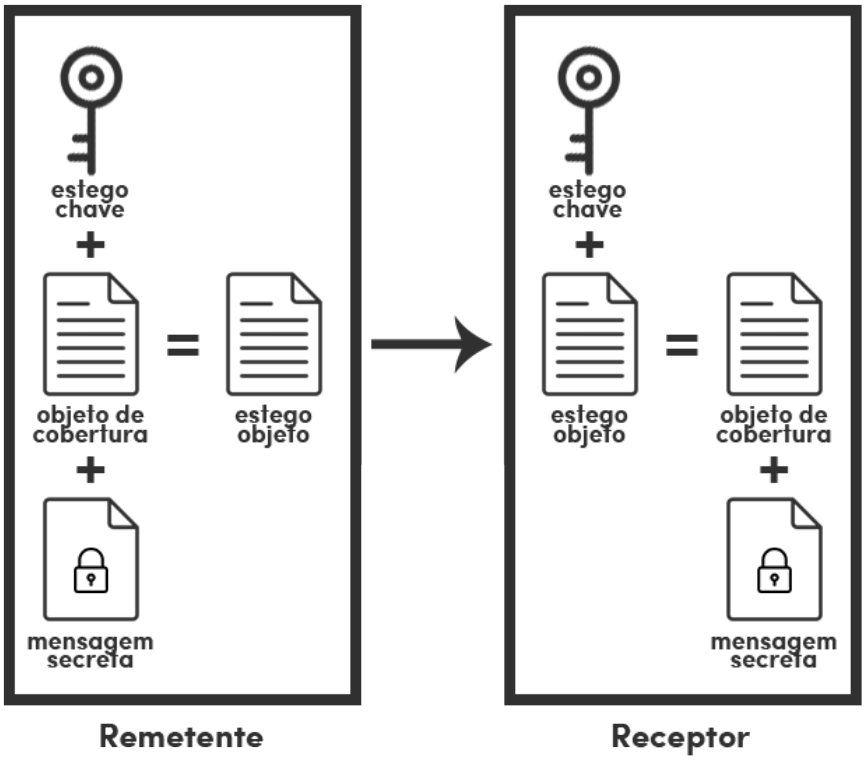
2.1. Conceitos Básicos

2.1.1. Esteganografia

Segundo Katzenbeisser (1999) podemos generalizar grande parte dos métodos esteganográficos em um princípio comum: o remetente deseja compartilhar uma mensagem secreta (m) com um receptor pré informado, para isso ele escolhe um **objeto de cobertura** (c) que possa ser visto como uma mensagem pública. Então, utilizando-se de algum método específico, o remetente insere m dentro de c. Pode-se ainda utilizar uma **estego-chave** (k) para controlar o processo de inserção de m. O resultado desse processo é chamado de **estego-objeto**, e deve ter uma estrutura que permita a extração de m desde que k (caso exista) seja conhecido, sem que seja necessário conhecer o c original. Uma mesma cobertura não deve ser utilizada duas vezes, já que um atacante em posse de duas versões de c pode utilizar as divergências entre elas para detectar a existência de uma

mensagem e possivelmente recuperar *m* [Katzenbeisser 1999]. Tal esquema está ilustrado na Figura 1.

Figure 1. Diagrama geral de um método esteganográfico.



Em tempos anteriores à popularização de computadores a esteganografia possuía um leque de aplicações fortemente confinado à transmissão de mensagens secretas entre dois correspondentes. A razão disso passa primeiramente pela falta de eficiência nos métodos utilizados, e em seguida pelo fato de que as aplicações modernas que conhecemos hoje sequer eram um assunto existente na sociedade da época.

O campo de marcas d'água digitais surge com a necessidade de inserir e verificar provas de autoria e propriedade de recursos digitais, e avança conforme as legislações específicas de grandes países ou blocos são sancionadas. Entretanto, nessa área os requisitos para a utilização de um algoritmo diferem dos habituais, dado que existe a necessidade de impedir que a marca seja retirada ou corrompida por transformações no arquivo. Além disso, a invisibilidade completa da mensagem nem sempre é indispensável e podemos ter marcas d'água visíveis para um observador externo. Mesmo com essas ressalvas, é notório que os últimos anos de evolução das técnicas de esteganografia foram fortemente voltados para aplicações na área de marcas d´água digitais, por seu apelo comercial latente [Evsutin et al. 2020].

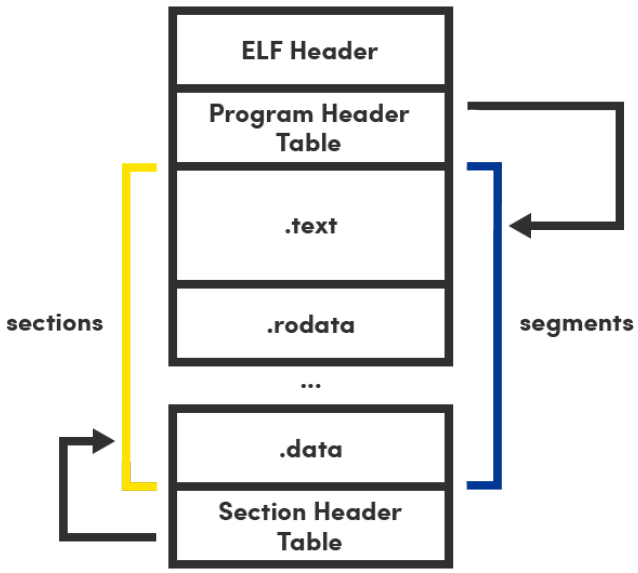
2.1.2. Arquivos ELF

O formato ELF foi desenvolvido inicialmente pela Unix System Laboratories em 1988, ganhando notoriedade e implementações em outros sistemas, até que em 1999 foi adotado como padrão pelo sistema Unix e seus derivados. Por ser facilmente generalizado para diversos tipos de sistemas e possuir suporte para a maioria das funcionalidades utilizadas pela indústria atualmente, continua até hoje figurando entre os formatos mais relevantes para arquivos executáveis.

Estruturalmente, um arquivo ELF é dividido em duas estruturas principais: segmentos e seções. Os segmentos contém informações úteis para o programa durante seu tempo de execução, como o local e valores de constantes definidas em código e as próprias instruções do programa. As seções possuem dados utilizados em tempo de ligação ou para *debugging* como símbolos gerados no processo de compilação e endereços de realocação de memória [elf 1995].

Para organizar essas informações, é utilizado um esquema de cabeçalhos contendo metadados e posições dos segmentos e seções. Logo no início do arquivo está o *ELF Header* detalhando informações como arquitetura alvo, tipo de arquivo e posições para os próximos cabeçalhos. Em seguida começam os *Program Headers* que definem como estão dispostos os segmentos no arquivo, os próprios segmentos e seções com seus conteúdos e por fim os *Section Headers* contendo as informações das seções. A estrutura típica de um executável contendo código e dados pode ser vista na Figura 2, notando que é comum essa sobreposição entre secções e segmentos na memória [elf 1995].

Figure 2. Estrutura de um arquivo ELF contendo dados e código.



2.1.3. Arquitetura RISC-V

Assim como nem todos os humanos compreendem a mesma língua, nem todas as máquinas sabem interpretar a mesma linguagem. Existem diversas formas de representar a linguagem de máquina desenvolvidas ao passar das décadas por diferentes fabricantes de computadores, cada máquina possuindo seu próprio conjunto de instruções que é capaz de executar. Esse conjunto é comumente denominado ISA e todas as máquinas que compreendem um mesmo conjunto são ditas da mesma arquitetura.

Atualmente duas grandes filosofias de construção de arquiteturas disputam espaço no mercado, CISC e RISC, tendo como mote principal a complexidade do seu conjunto de instruções. Enquanto máquinas CISC possuem um conjunto de instruções maior e essas instruções também contém um grau de complexidade mais elevado, as máquinas RISC possuem poucas operações e buscam realizar as operações de forma fragmentada em uma quantidade maior de passos simples [Patterson and Sequin 1981].

Em consequência dos princípios básicos de cada tipo de máquina, é possível inferir características que serão comuns a todas as arquiteturas de cada classe. Máquinas CISC costumam gerar códigos menores, pois aglutinam diversas operações em uma só instrução, enquanto nas implementações RISC temos o oposto. Entretanto, apesar da quantidade total de instruções geradas para um mesmo código na máquina RISC ser maior, cada instrução possui uma execução mais rápida devido a sua simplicidade, retomando de volta a performance perdida por executar um número maior de instruções [Patterson and Sequin 1981].

Nos dias atuais a mais popular representação de uma arquitetura CISC é a x86, desenvolvida inicialmente pela Intel em 1978 como uma extensão da arquitetura 8080 [Asanović and Patterson 2014]. Apesar de ser coloquialmente referida como uma arquitetura, é na verdade um conjunto que abarca diversas ISAs desenvolvidas ao longo das décadas. Para máquinas RISC podem ser citadas as arquiteturas ARM e RISC-V, sendo que a última possui esse nome por ser a quinta geração de arquiteturas RISC a ser desenvolvida pela Universidade da Califórnia [Patterson and Hennessy 2017].

A arquitetura RISC-V é uma ISA desenvolvida de forma aberta e sem cobrança de *royalties*, originalmente pelo departamento de engenharia elétrica e ciências da computação da Universidade da Califórnia, campus Berkeley. Entretanto, desde o início em 2010 até o presente, muitos contribuidores externos, independentes ou em contribuições corporativas, participaram do projeto. Diferentemente do usual para projetos acadêmicos, o RISC-V busca não apenas ser um objeto de estudo teórico, mas também ter viabilidade para projetos reais na indústria [Asanović and Patterson 2014].

2.2. Revisão Bibliográfica

Apesar de não ser um assunto extensamente explorado, a esteganografia utilizando código executável já foi proposta e estudada por outros autores ao longo dos últimas décadas. O projeto Hydan utiliza instruções semânticamente equivalentes como decisores binários para inserir dados dentro de um código executável para a arquitetura x86. Entretanto, a razão de codificação de em média 1 bit de mensagem para cada 100 bits do objeto de cobertura encontrada é insatisfatória para alguns casos de uso [El-Khalil and Keromytis 2004]. No mesmo ano, [Anckaert et al. 2004] expandem o con-

ceito inicial do Hydan, realizando um estudo sobre a redundância em programas da arquitetura x86 e formulando critérios de substituição claros. Além disso, esse trabalho também traz um *framework* para análise de detectabilidade dos estego-objetos gerados.

O *whitepaper* publicado por [Wright 2008] desenvolve em cima de um tópico já citado pelos autores originais do Hydan: a fragilidade do trabalho em relação a detectabilidade. O método aplicado pelos pesquisadores não leva em consideração a distribuição estatística das instruções em cada plataforma, permitindo que uma análise seja feita buscando anomalias, comparando instruções contidas no programa alvo com a distribuição normalmente encontrada em programas compilados para a mesma plataforma.

Outra abordagem foi tomada por [Shin et al. 2008], que desenvolve um método para inserção de dados em arquivos do formato PE, adicionando os dados na seção *.text* do binário e tomando providências de realocação caso seja necessário. Apesar desse método possuir uma flexibilidade maior e capacidade de codificação virtualmente infinita, ele altera o tamanho do arquivo em disco, portanto pode ser detectado por um observador externo em posse do objeto de cobertura, invalidando essa técnica para aplicações em que o arquivo original seja de alguma forma público. De forma complementar, um trabalho posterior propõe um método similar cobrindo as mesmas plataformas e adicionando uma camada de criptografia [Zaidan et al. 2009].

Além de métodos diferentes para a inserção dos dados no objeto de cobertura, a natureza dos próprios dados é espaço de pesquisas. [Lu et al. 2014] elaboraram a ideia de utilizar esteganografia não só para esconder uma mensagem dentro de um código fonte, mas sim para esconder parte do código fonte dentro de si mesmo. Os autores utilizam ROP em combinação com esteganografia para que certos conjuntos de instrução sejam invisíveis à analisadores estáticos de código mas ainda sejam executadas em tempo de execução. Como prova de conceito é apresentada a ferramenta RopSteg que utiliza uma versão modificada do algoritmo de *Galileo* para gerar segmentos de código contendo estruturas de retorno que não estavam no binário original. Entretanto, apesar de conseguirem êxito para fazer as alterações no arquivo, devido à ROP ser frequentemente associada com malwares, os autores alertam que existe a possibilidade do arquivo gerado ser marcado como malicioso por sistemas de detecção.

Por fim, existem ainda implementações não acadêmicas porém relevantes, utilizando propostas levantadas pelos autores anteriores e que de certa formam fornecem alguma validação da viabilidade material desses métodos. A ferramenta *steg86* apropria-se da ideia inicial dos autores do Hydan para desenvolver uma interface de linha de comando simples para ocultação de dados em binários compilados para as plataformas x86 e AMD64, oferecendo suporte tanto para ELF quanto para PE. Além de operações básicas de injeção e extração de mensagens, o utilitário também oferece um comando para fazer uma análise de um binário e determinar quantos bits podem ser inseridos nele [Woodruff 2020].

3. Desenvolvimento

3.1. Substituição de Instruções Equivalentes

A estratégia de utilizar escolhas dentro de uma única instrução como estrutura de codificação binária é amplamente utilizada por autores desde os primeiros trabalhos realizados na área, [El-Khalil and Keromytis 2004] baseiam sua solução em criar tabelas de

instruções equivalentes e atribuir um valor para cada alternativa, dessa forma para cada instrução pertencente a um conjunto de instruções equivalentes, podemos codificar $\log_2 n$ bits, sendo n o tamanho do conjunto. Os autores catalogaram 18 conjuntos de equivalência dentro da ISA `x86`, variando entre operações aritméticas e binárias, com equivalências produzidas por ordem de operandos e inversas matemáticas de instruções.

Devido as diferenças filosóficas entre arquiteturas RISC e CISC, a aplicação direta dessa técnica possui uma eficiência reduzida, com um número menor de conjuntos equivalentes a serem explorados por um algoritmo de substituição. Muitas das redundâncias comuns em outras ISAs foram removidas propositalmente durante o desenvolvimento do RISC-V, restando apenas aquelas que seriam impossíveis de remover como as equivalências lógicas e aritméticas axiomáticas. Dado isso, foram buscadas instruções com propriedades comutativas no conjunto base de instruções, resultando em cinco grupos contendo exatamente duas instruções cada que estão detalhadas na Tabela 1. Por termos apenas duas instruções por grupo, é possível codificar $\log_2 2$ bits por cada ocorrência no programa.

Nome do grupo	Instrução
add-group	add rd, rs1, rs2 add rd, rs2, rs1
and-group	and, rd, rs1, rs2 and, rd, rs2, rs1
or-group	or, rd, rs1, rs2 or, rd, rs2, rs1
beq-group	beq, rd, rs1, rs2 beq, rd, rs2, rs1
bne-group	bne, rd, rs1, rs2 bne, rd, rs2, rs1

Table 1. Conjuntos de instruções equivalentes no RISC-V

As instruções encontradas são operações comuns e presentes em todo programa de computador, o que indica uma possível abundância de espaço para codificação de informação. Ademais, não existe uma convenção da arquitetura que denote qual deva ser a ordem dos operandos de uma determinada instrução, dessa forma podemos descrever o passo a passo para codificar bits utilizando essa estratégia utilizando o Algoritmo 1.

Inicialmente define-se *P* como o conjunto contendo todas as instruções do programa alvo ordenadas pelo seu endereço relativo ao arquivo, *I* como o conjunto de instruções capazes de codificar um bit (vistos na Tabela 1) e *M* como a mensagem binária a ser inserida no arquivo. Foi definido que para esta implementação o critério de final da mensagem é dado por um caractere ASCII nulo, representado por 8 bits de valor zero. Em seguida itera-se por cada caractere da mensagem buscando a próxima instrução que esteja contida em *I*. Ao encontrar uma instrução candidata, caso seus dois operandos sejam idênticos é necessário buscar outra pois é impossível codificar uma escolha nessa situação. Do contrário, é possível codificar um bit ordenando os operandos de acordo com um critério pré estabelecido, durante a execução deste trabalho usaremos como critério a função f definida por:

```
1  P ← {...};
2  I ← {...};
3  M ← "01000110010011000100000100000000";
4  message_index ← 0;
5  while message_index ≤ |M| do
6      encoded ← false;
7      while encoded = false do
8          instruction ← next(P);
9          if instruction not in I then
10             continue;
11         end
12         if instruction.rs1 = instruction.rs2 then
13             continue;
14         end
15         if char = 1 then
16             if not (instruction.rs1 ¿ instruction.rs2) then
17                 swap(instruction.rs1, instruction.rs2);
18             end
19         else
20             if not (instruction.rs1 ¡ instruction.rs2) then
21                 swap(instruction.rs1, instruction.rs2);
22             end
23         end
24         encoded ← true;
25     end
26 end
27 return P;
```

Algoritmo 1: Codificação de bits utilizando substituição de instruções equivalentes

$$f(rs1, rs2) = \begin{cases} 1 & \text{se } rs1 > rs2 \\ 0 & \text{se } rs2 > rs1 \\ \phi & \text{caso contrário} \end{cases} \quad (1)$$

Ao final da iteração pelos caracteres da mensagem o conjunto *P* representa a lista de instruções já com a mensagem embutida em si, a partir desse ponto o codificador RISC-V pode aplicar o processo reverso para transformar as instruções em linguagem de montagem para código de máquina e escrever o arquivo final que representa o estego-objeto.

```
1  P ← {...};
2  I ← {...};
3  M ← string();
4  last_char ← string() ;
5  while last_char ≠ "00000000" do
6  |  if last_char.length = 8 then
7  |  |  last_char ← string() ;
8  |  end
9  |  instruction ← next(P);
10 |  if instruction not in I then
11 |  |  continue;
12 |  end
13 |  if instruction.rs1 = instruction.rs2 then
14 |  |  continue;
15 |  end
16 |  if instruction.rs1 ¿ instruction.rs2 then
17 |  |  M += "1";
18 |  else
19 |  |  M += "0";
20 |  end
21 end
22 return M;
```

Algoritmo 2: Decodificação de bits utilizando substituição de instruções equivalentes

Para decodificar uma mensagem propõe-se um algoritmo semelhante ao anterior, tomando *P* como o conjunto de todas as instruções do programa, *I* como o conjunto de instruções codificáveis e *M* como uma *string* vazia que ao final da execução receberá a mensagem decodificada. Entretanto, após encontrar uma instrução candidata para a técnica um bit é extraído a partir da ordem de seus operandos e a concatenação final de todos os bits contém a mensagem, respeitando o critério de parada de caractere nulo

controlado por uma variável que é verificada a cada 8 bits para uma representação em ASCII.

3.2. Aplicação Desenvolvida

Para todos os desenvolvimentos práticos foi definido o uso da linguagem Python, conhecida por sua simplicidade e clareza de código, o que facilita a compreensão para os pesquisadores e desenvolvedores de diversas áreas, em sua versão 3.10. Além disso, a vasta gama de bibliotecas e módulos disponíveis em Python, como `struct`, `pickle`, e `array`, oferece funcionalidades prontas para o tratamento de dados binários, economizando tempo e esforço no desenvolvimento. A portabilidade do Python também é uma vantagem, já que a linguagem é suportada em diferentes sistemas operacionais, garantindo que os resultados do projeto sejam facilmente reproduzíveis. Por fim, a comunidade Python ativa e os recursos de documentação tornam mais simples a resolução de problemas e o suporte durante o desenvolvimento do projeto.

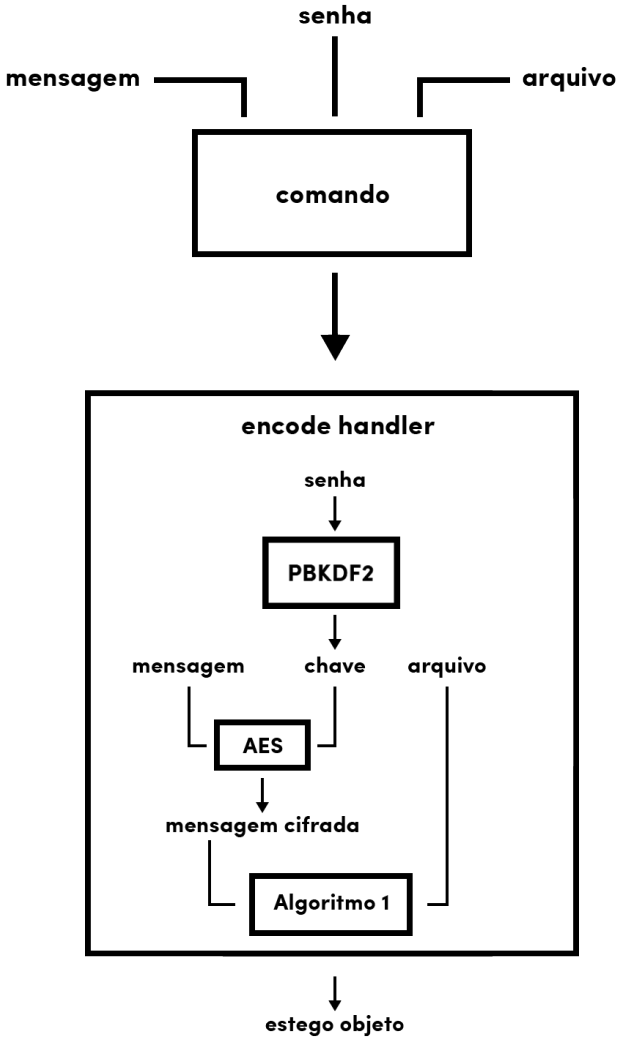
3.2.1. Comando `encode`

O comando `encode` é responsável por receber uma mensagem em texto plano, criptografá-la e inserir-la no arquivo de fonte, gerando ao final o estego-objeto. Para os propósitos deste trabalho foi utilizada criptografia simétrica com o algoritmo AES sendo instanciado em *Output Feedback Mode*, uma modalidade do AES que trabalha de forma serial utilizando o vetor de inicialização para gerar o texto cifrado de forma iterativa, com uma operação *XOR* entre o texto e um bloco cifrado a partir da chave, gerando uma distribuição de probabilidade pseudoaleatória para cada bit da mensagem cifrada final [Dworkin 2001]. Todas as chaves são derivadas a partir de uma senha escolhida pelo usuário utilizando PBKDF2, uma função computacional capaz de derivar chaves pseudoaleatórias a partir de uma entrada em texto plano, utilizando como função pseudoaleatória HMAC em sua forma SHA512, com 2^{15} iterações e um *salt* de 16 bytes [Turan et al. 2010].

Após criptografada a mensagem é concatenada ao *salt*, codificada em *base64* e inserida conforme definido no Algoritmo 1, adicionando um caractere de final de mensagem para delimitar o tamanho total do texto inserido e utilizar como critério de parada no processo de decodificação. Nesse passo foi realizada uma otimização de implementação no algoritmo, o parâmetro de entrada *P* é fornecido a partir de uma função do tipo *generator*, que age como um objeto iterável retornando a cada iteração uma instrução do programa. Dessa forma, mesmo que seja utilizado um arquivo de tamanho elevado as instruções são trazidas para memória apenas quando serão utilizadas, resultando em diminuição drástica do uso de recursos para execuções em grandes arquivos.

O arquivo modificado é salvo em caminho alternativo ao original, a partir de uma cópia do original, alterando apenas as instruções codificadas no programa o que diminui o número de operações em disco realizadas pela implementação, dessa forma o usuário mantém o objeto de cobertura intacto e recebe um estego objeto recém criado. Caso a mensagem exceda a capacidade de codificação do arquivo informado é levantada uma exceção que alerta o usuário sobre o ocorrido e o arquivo não é alterado em nenhum aspecto. O funcionamento completo do comando de codificação está ilustrado na Figura 3.

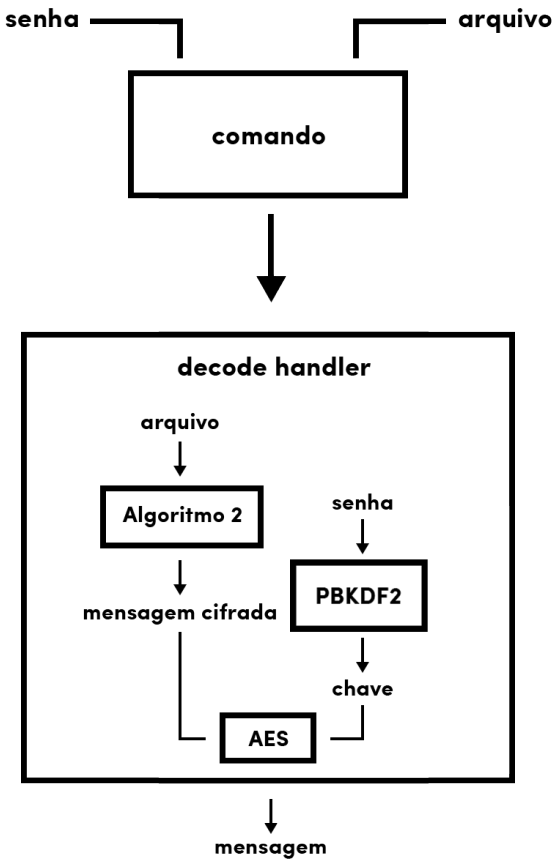
Figure 3. Esquemático do funcionamento do processo de codificação.



3.2.2. Comando decode

O processo de decodificação é realizado de forma inversa a codificação, recebendo um arquivo binário que se presume ser um estego objeto criado com a mesma técnica e uma senha para descriptografar a mensagem. Inicialmente buscamos toda a mensagem criptografada contida no arquivo, utilizando o Algoritmo 2 como base até encontrar um caractere de parada, caso nenhum caractere de parada seja encontrado a aplicação levanta uma exceção alertando o usuário de que o arquivo não contém dados relevantes inseridos em si. Ao encontrar uma mensagem geramos uma chave a partir da senha utilizando PBKDF2 e o *salt* contido no início da mensagem, ao ponto em que podemos descriptografar a mensagem utilizando o AES com a chave derivada e retornar a mensagem em texto plano para o usuário. O funcionamento completo do comando de codificação está ilustrado na Figura 4.

Figure 4. Esquemático do funcionamento do processo de decodificação.



4. Resultados

Para avaliar os resultados obtidos pelas técnicas de esteganografia e validar o ferramental de apoio desenvolvido durante este trabalho foram utilizado dois conjuntos distintos de programas como objetos de experimento. O primeiro conjunto contém a compilação de 2 sistemas operacionais, o segundo conjunto é composto 22 programas utilitários de bancos de dados. Os sistemas operacionais foram selecionados devido a alta gama de operações distintas realizadas por esse tipo de programa durante sua execução. Os utilitários com enfoque para bancos de dados foram escolhidos por possuirem em seu código algoritmos variados de ordenação, busca e transformação de dados, com o intuito de entender se um programa altamente focado em operações em disco é mais suscetível a técnicas de esteganografia.

Para ambos os conjuntos de programas foi utilizado o processo de compilação cruzada a partir de uma máquina x86 tendo como arquitetura alvo o RISC-V para eliminar a necessidade de possuir uma placa física implementando RISC-V. No primeiro conjunto os softwares utilizados foram o EPOS, desenvolvido com o propósito de ser um SO enxuto e eficiente capaz de ser executado em microprocessadores de sistemas embarcados e o Linux, um SO de propósito geral utilizado na maioria dos dispositivos pelo mundo e que conta com funcionalidades mais variadas e robustas do que o primeiro. Para o segundo conjunto foram utilizados os utilitários gerados na compilação do SGBD PostgreSQL, que

incluem o próprio SGBD e outros módulos utilizados para funcionalidades específicas do software.

4.1. Validação Quantitativa

Após a elaboração do algoritmo apresentado para codificação utilizando substituição de instruções equivalentes foi realizado um experimento com a motivação de examinar a quantidade de bits que poderiam ser codificados dentro de cada arquivo presente no conjunto de testes. Para isso uma pequena modificação foi realizada no algoritmo, incrementando um contador sempre que encontrada uma instrução explorável pela técnica de modo a obter no final o total de bits codificáveis. Foi possível verificar que a técnica possui uma capacidade de codificação baixa quando comparada com os resultados obtidos para a arquitetura x86 por outros autores. A taxa de codificação média obtida por essa implementação foi de 0,03%, inferior aos 0,9% demonstrados por [El-Khalil and Keromytis 2004] e os 3,7% apresentados por [Anckaert et al. 2004]. Os resultados inferiores para a arquitetura alvo desse trabalho são explicados principalmente pelo tamanho dos conjuntos de substituições disponíveis, enquanto a arquitetura da Intel conta com 18 conjuntos contendo entre 2 a 5 instruções, o RISC-V apresenta apenas 5 conjuntos de 2 instruções cada, reduzindo tanto a frequência de aparição de instruções exploráveis no programa quanto a quantidade de bits codificados por instrução.

Entretanto, apesar de não apresentar resultados satisfatórios para codificação de grandes quantidades de dados o experimento comprova a viabilidade do uso dessa técnica para inserção de uma quantidade reduzida de bits, possibilitando o uso do método para aplicações específicas que não tenham como requisito o uso de mensagens extensas.

4.2. Validação Qualitativa

Existem ataques comuns a classe de técnica utilizada na implementação deste trabalho descritos na litetura, que buscam encontrar assinaturas ou características comuns à transformações de código aplicadas [Anckaert et al. 2004]. Um atacante buscando detectar essas assinaturas pode buscar pela presença de instruções pouco utilizadas em uma arquitetura a fim de identificar um padrão de substituição. Entretanto como o Algoritmo 1 não substitui instruções, trabalhando apenas com substituição de operandos, não há risco de ter essa propriedade explorada. Outra possibilidade é a detecção de uma frequência anormal de um certo tipo de instruções, também descartada pelo motivo anterior. Por fim, o atacante pode inspecionar o comportamento dos *jumps* entre endereços do programa, buscando valores discrepantes que não seria naturalmente endereçados por compiladores. Tal vulnerabilidade também é inefetiva contra a aplicação desenvolvida pela ausência da alteração de valores imediatos em instruções de controle.

Nota-se então que os ataques já observados na bibliografia disponível não tem a capacidade de detectar as transformações realizadas pelo algoritmo proposto, o que não significa dizer que o mesmo seja indetectável, considerando que os ataques foram planejados para expor vulnerabilidades em técnicas aplicadas na arquitetura *x86* que possui uma gama de possibilidades de alteração superior. Em fato, a eficiência de codificação de instruções da arquitetura RISC-V simultâneamente favorece a avaliação qualitativa de transformações binárias enquanto desfavorece uma avaliação quantitativa, reduzindo o número de permutações equivalentes dentro do programa e tornando a codificação de informações mais silenciosa porém menos eficiente.

5. Conclusão

O algoritmo proposto é capaz de explorar redundâncias provenientes de operações comutativas na arquitetura RISC-V para codificar um bit a cada operação suportada e sua implementação foi desenvolvida de forma a minimizar o uso de recursos para possibilitar a execução eficiente mesmo em arquivos de grande porte. Apesar do resultado positivo na viabilidade da aplicação é necessário ressaltar que sua eficácia é reduzida em comparação com as técnicas semelhantes apresentadas por [El-Khalil and Keromytis 2004] e por [Anckaert et al. 2004], ambas implementadas em arquiteturas CISC, que favorecem as técnicas por possuírem uma codificação de instruções com redundância elevada.

Ao avaliar a técnica por outro prisma, relevando a baixa taxa de codificação e focando primariamente na detectabilidade e corretude quanto ao problema dos prisioneiros, entende-se que os pontos negativos podem ser mitigados com o uso do algoritmo em situações específicas, aproveitando-se da dificuldade em revelar a mensagem sem conhecimento da chave para inserir mensagens de tamanho reduzido em arquivos. Tais situações podem ser exploradas para embutir assinaturas e identificadores em *softwares* distribuídos comercialmente com o objetivo de rastrear violações de direitos autorais no combate à pirataria.

6. Trabalhos Futuros

A execução deste trabalho revela algumas portas em aberto para a evolução das técnicas de esteganografia em código compilado, principalmente considerando a arquitetura RISC-V. Durante a revisão bibliográfica foram encontrados pontos de exploração com potencial para codificação de dados, que poderiam ser unidos a técnica desenvolvida para incrementar a taxa de codificação. Inicialmente existe a possibilidade de explorar as instruções `HINT`, reservadas no espaço de endereçamento da arquitetura como livres para uso de implementações, normalmente utilizadas para otimizações de performance mas que poderiam ser subvertidas para carregar bits de mensagens. Além disso, no decorrer da pesquisa realizada o foco foi aplicar transformações diretamente no conjunto de instruções base da arquitetura, deixando de fora diversas extensões de propósitos específicos que apesar de possuírem frequência menor de aparecimento em um programa arbirtrário podem aumentar a taxa total de codificação de dados.

Ainda no escopo da arquitetura RISC-V e da implementação atual, levando em consideração que a validação de detectabilidade da técnica foi realizada a partir de ataques comuns direcionados a binários compilados para *x86*, é necessário expor a técnica à uma gama maior de ataques que possam expor vulnerabilidades específicas da arquitetura para reforçar a confiabilidade do algoritmo.

Por fim, existem outras arquiteturas RISC amplamente utilizadas em dispositivos ao redor do mundo, principalmente da família ARM nas quais os conhecimentos adquiridos podem ser aplicados, resultando na portabilidade da técnica para um leque maior de usuários e possivelmente em resultados superiores se considerarmos que o endereçamento do RISC-V possui uma maior rigidez aos princípios RISC.

References

(1995). Tool Interface Standard (TIS) executable and linking format (elf) specification. `https://refspecs.linuxbase.org/elf/elf.pdf`. Acessado em

17/06/2023.

Anckaert, B., De Sutter, B., Chanet, D., and De Bosschere, K. (2004). Steganography for executables and code transformation signatures. In *17th International Conference on Information Security and Cryptology ICISC 2004*, number 3506 in LNCS, pages 425–439.

Asanović, K. and Patterson, D. A. (2014). Instruction sets should be free: The case for risc-v. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley.

Dworkin, M. J. (2001). Recommendation for block cipher modes of operation :. Technical report.

El-Khalil, R. and Keromytis, A. D. (2004). Hydan: Hiding information in program binaries. In Lopez, J., Qing, S., and Okamoto, E., editors, *Information and Communications Security*, pages 187–199, Berlin, Heidelberg. Springer Berlin Heidelberg.

Evsutin, O., Melman, A., and Meshcheryakov, R. (2020). Digital steganography and watermarking for digital images: A review of current research directions. *IEEE Access*, 8:166589–166611.

Johnson, N. F. and Jajodia, S. (1998). Exploring steganography: Seeing the unseen. *Computer*, 31(2):26–34.

Katzenbeisser, S. (1999). *Information Hiding Techniques for Steganography and Digital Watermarking*. Artech House Print on Demand.

Lu, K., Xiong, S., and Gao, D. (2014). Ropsteg: Program steganography with return oriented programming. CODASPY '14, page 265–272, New York, NY, USA. Association for Computing Machinery.

Mahoney, W., Franco, J., Hoff, G., and McDonald, J. T. (2018). Leave it to weaver. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop*, SSPREW-8, New York, NY, USA. Association for Computing Machinery.

Patterson, D. A. and Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Patterson, D. A. and Sequin, C. H. (1981). Risc i: A reduced instruction set vlsi computer. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, page 443–457, Washington, DC, USA. IEEE Computer Society Press.

Shin, D., Kim, Y., Byun, K., and Lee, S. (2008). Data hiding in windows executable files. *Australian Digital Forensics Conference*.

Turan, M. S., Barker, E. B., Burr, W. E., and Chen, L. (2010). Recommendation for password-based key derivation :. Technical report.

Woodruff, W. (2020). steg86. `https://github.com/woodruffw/steg86`.

Wright, C. (2008). Detecting hydan: Statistical methods for classifying the use of hydan based stegonagraphy in executable files.

Zaidan, A., Bahaa, B., and Othman, F. (2009). New technique of hidden data in pe-file with in unused area one. *International Journal of Computer and Electrical Engineering*.