UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
PROGRAMA DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Mateus Manoel Pereira

IMPLEMENTAÇÃO DE UM SISTEMA ANTI-RANSOMWARE COM ABORDAGEM DE DETECÇÃO DE CIFRAGEM EM TEMPO REAL

Florianópolis, Santa Catarina – Brasil
2022

Mateus Manoel Pereira

IMPLEMENTAÇÃO DE UM SISTEMA ANTI-RANSOMWARE COM ABORDAGEM DE DETECÇÃO DE CIFRAGEM EM TEMPO REAL

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Jean Everson Martina

Florianópolis, Santa Catarina – Brasil
2022

Este TCC é inteiramente dedicado ao meu irmão de coração, Roger Albino.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

Considerado a ameaça número um de vírus, os ataques de *ransomware* vêm crescendo muito com o passar dos anos. Segundo uma pesquisa desenvolvida pela empresa de segurança virtual *ExtraHop* (CANALTECH, 2022), ao menos 85% dos entrevistados afirmaram ter sofrido pelo menos um ataque de vírus do tipo *ransomware* por ano. Ainda, 72% das empresas acabam pagando a quantidade de dinheiro exigida pelos cibercriminosos. Em sua forma mais comum, o *ransomware* criptografa os arquivos da vítima mantendo-os inacessíveis até o pagamento de um resgate, geralmente em criptomoeda, afetando desde usuários comuns até grandes empresas. Essas por sua vez tendem a pagar o resgate para terem seus arquivos de volta, gerando gastos na casa de milhões e contribuindo com organizações criminosas (ZDNET, 2021). Em Junho de 2021, a maior produtora de carne do mundo, a JBS, teve três de seus matadouros paralisados devido a um *ransomware* e 11 milhões de dólares foram pagos aos criminosos (TECHSPOT, 2021).

Um dos motivos para a popularização do *ransomware* foi o surgimento do *Ransomware as a Service* (RaaS) na *Dark Web* (CFO DIVE, 2022), com isso o vírus é disponibilizado para o comprador e este tem o trabalho apenas de lançá-lo na rede e esperar que as vítimas paguem o resgate.

Livrar-se de uma ameaça como essa requer muita cautela com todos os ativos que se deseja proteger. Manter todos os dispositivos sempre atualizados, fazer backup constantemente e estabelecer uma política de segurança com os funcionários estão entre as principais medidas de prevenção contra o *malware*. Além disso, existem antivírus capazes de identificar que uma determinada aplicação está tentando executar um *ransomware* através da correlação de assinatura do software com *malwares* já identificados, entretanto esta técnica falha em detectar *ransomwares* mais novos e nunca antes usados (ASLAN; SAMET, 2020).

1.1 O QUE É *RANSOMWARE*?

Ransomware é um termo muito amplo que em sua essência define uma classe de vírus de computador que tem como objetivo extorquir a vítima digitalmente. Pode-se dividir o *ransomware* em dois tipos, os que criptografam, ofuscam ou impedem a vítima de acessar seus dados e os que restringem ou bloqueiam o acesso ao sistema operacional. Dentro desses dois tipos ainda existem subdivisões de tipos que levam em conta a semelhança na engenharia do ataque. É importante ressaltar que os ataques *ransomware* não são limitados a uma plataforma específica, existindo vírus dessa classe para vários dispositivos. Nenhum sistema operacional está fora de perigo.

Neste trabalho o foco será os *ransomwares* do tipo que criptografam os arquivos da vítima impedindo-a de acessá-los, conhecidos como *Crypto Ransomwares*. A moti-

vação para essa escolha se dá pelo risco notadamente alto e popularidade que está tomando. No ano de 2022 o *Crypto Ransomware* é a forma mais comum de ataque de extorsão digital (NETAPP, 2022).

1.1.1 As fases de um ataque *Ransomware*

Allan e Timothy apresentam a anatomia de um ataque *ransomware* como um processo que pode ser dividido em cinco partes: implantação, instalação, comando e controle, destruição e extorsão (LISKA; GALLO, 2017).

Na fase de implantação todos os arquivos necessários para a execução do ataque são baixados na máquina da vítima. Estes arquivos variam de ataque para ataque, mas os mais comuns estão relacionados diretamente a criptografia dos ativos da vítima e permanência do vírus no sistema. Existem várias formas de fazer com que esses arquivos maliciosos cheguem ao computador da vítima, engenharia social e exploração de vulnerabilidades do próprio sistema são as mais comuns.

A segunda fase do processo é a instalação, essa fase tem como objetivo infectar a máquina da vítima e passar despercebido pelas técnicas de segurança do sistema, no *Windows*, por exemplo, o vírus pode se aproveitar dos registros do sistema para automatizar a inicialização do *malware* junto com o computador. Além das vulnerabilidades do sistema também é possível que o vírus se aproveite de falhas de aplicações instaladas, um dos grandes exemplos de aplicações que permitiam essa prática é o *Flash* da *Adobe* que não só foi descontinuado mas também teve seu uso bloqueado devido a grande presença de vulnerabilidades exploradas (BLEEPINGCOMPUTER, 2020).

Um *ransomware* sofisticado funciona como uma programa cliente-servidor, sendo os clientes as vítimas infectadas. Na etapa de comando e controle são coletadas diversas informações do sistema da vítima a fim de instruir o vírus a se comportar de maneira passiva ou intrusiva, além disso, essa informações também podem servir para o atacante determinar que arquivos serão criptografados e qual será o valor pago no resgate dos dados. Muitas vezes os *ransomwares* ficam semanas coletando informações antes de agir no computador da vítima.

Quando o *malware* recebe o comando para agir no dispositivo começa a fase de destruição, nela todos os arquivos selecionados na etapa anterior começam a ser criptografados. Quando todos os arquivos não estão mais acessíveis para o usuário infectado temos o estágio de extorsão. Uma tela explicando à vítima o que está acontecendo é exibida, nela também estão as informações necessárias para o pagamento do resgate. Na maioria dos ataques, o valor da extorsão é mais barato que os custos para recuperar ou reconstruir os dados incentivando o pagamento do resgate. Este valor já foi calculado na fase de comando e controle.

1.2 OBJETIVOS

1.2.1 Objetivos Gerais

O objetivo deste trabalho é propor um software de prevenção e detecção de *ransomware* em ambientes Linux capaz de neutralizar ataques dia zero antes que causem danos severos à máquina do usuário. O software também tem como objetivo ser capaz de realizar suas operações sem deteriorar a usabilidade da máquina, sendo o mais discreto possível.

1.2.2 Objetivos Específicos

O trabalho conta com os seguintes objetivos específicos: desenvolver um módulo de *kernel* para o Linux, detectar ameaças do tipo *ransomware* e desenvolver um programa de prova para testar a solução proposta.

1.2.3 Justificativa

Os ataques de *malwares* contra ambientes Linux tiveram um aumento significante em 2017 (SC MAGAZINE, 2017) e continuam crescendo. Além disso, hoje em dia a maioria dos servidores rodam Linux. É de extrema importância proteger sistemas de uma ataque que pode interromper todas as funcionalidades não essenciais de um SO em poucos minutos. Outro ponto a se destacar é que a maioria das soluções, analisadas pelo autor, contra *ransomwares* que existem atualmente falham em construir algo transparente ao usuário do sistema, já em casos onde isso ocorre, como em (JOSHI *et al.*, 2021), a necessidade de ter uma lista de processos permitidos, como o *explorer.exe* do *Windows*, para que o usuário possa fazer operações em seus arquivos abre brechas para um atacante mal intencionado.

1.3 PRIMEIROS PASSOS

Com a finalidade de alcançar os objetivos expostos nessa proposta foi percorrido um longo caminho. No começo do desenvolvimento deste projeto, muito foi estudado na literatura e nas práticas que dizem respeito a área sendo pesquisada, foi formulado hipóteses que tenham por finalidade contribuir com a detecção de *ransomwares* em ambientes Linux, além de testes para obter-se resultados sobre as hipóteses. Tudo isso documentado nas próximas seções.

No que diz respeito ao conhecimento de tecnologias para implementação de *softwares/scripts* com a finalidade de testar e obter resultados para o projeto, novas ferramentas foram estudadas. Um exemplo disso é o sistema operacional Linux, conhecido por ser complexo e estar em constante desenvolvimento, trazendo novas funcionalidades. Aprimorar-se nele a nível de *kernel* foi um tarefa complicada que

demandou grande esforço e tempo.

Para a prova de conceito foi utilizado uma ferramenta de monitoramento de eventos Linux, *Fanotify*. Com ela é possível identificar eventos de leitura e escrita em arquivos e decidir se eles devem ocorrer ou serem bloqueados. Essa solução foi adotada para a prova de conceito já que o foco desta etapa era provar ser possível a detecção de criptografia de arquivos e a restauração de arquivos danificados.

2 REVISÃO BIBLIOGRÁFICA

Muito foi pesquisado à respeito de técnicas mais avançadas de prevenção e detecção de *ransomware* e foi notável que diferentes áreas tentam solucionar o problema de sua maneira. Melton tenta detectar um ataque *ransomware* através da análise de poder computacional, verificando o uso de um CPU Intel Core i7-4790 e um SSD. Os resultados do experimento mostram que é difícil distinguir entre certas ações realizadas pelo usuário e um ataque de encriptação. A precisão do software *anti-ransomware* aumenta conforme o tamanho do arquivo cresce chegando em 100% a partir dos 2,5GB, entretanto a solução também apresenta falso positivos (MELTON, 2018).

Em (POUDYAL; DASGUPTA, 2020) os autores utilizam aprendizado de máquina para detectar *ransomwares* em um ambiente de teste, vários algoritmos de classificação foram submetidos, parâmetros como chamadas a DLL e chamadas de sistema constituíram as regras de associação. Nos testes o melhor algoritmo classificador foi o *AdaBoost* com J48 (PANDEY; PRABHAKAR, 2016) com 99,54% de acurácia e apenas 0,005% de falso positivos. 1090 amostras de *ransomwares* foram usadas para compor o teste.

Outra abordagem na área de mineração de dados foi feita em (TAFKOV; MINCHEV, 2020), foi utilizada uma Rede Neural Residual Profunda (LU; GRAHAM; LI, 2019) treinada em detectar comportamentos de um ataque *ransomware*, como destruir pontos de restauração do sistema, estabelecer conexão http(s) para adquirir chave de criptografia e modificar arquivos em diretórios padrões. Com o experimento foi possível identificar 100% das amostras usadas no teste, incluindo alguns bem conhecidos como *WannaCry* e *Jigsaw*. O artigo não leva em consideração falso positivos e novamente, como em (POUDYAL; DASGUPTA, 2020), o experimento não traz nenhuma certeza quanto a eficácia em ataques contra um *ransomware* dia zero.

A *IBM Research* fez uma pesquisa (CONSTANTINESCU; SESHADRI, 2021) e levantou uma alternativa para detecção de *ransomware* em tempo real, com foco em proteção empresarial e escalabilidade. O artigo propõe uma análise no padrão de acesso a arquivos do dispositivo, seja ele *Windows*, Linux ou até mesmo no próprio IBM *Spectrum Scale*, um software de sistema de arquivos em *cluster* de alto desempenho criado pela IBM. Além da detecção de padrões, os autores sugerem também uma técnica de salvamento automático de versões de arquivos, para que, no caso de um ataque, o arquivo possa ser facilmente restaurado para uma versão não corrompida. Apesar do comprometimento dos autores com o objetivo de construir um mecanismo de detecção de *ransomware* com baixa frequência de falso positivos e falso negativos próximos de zero nenhum resultado foi apresentado.

Com a Entropia de Shannon é possível identificar a randomicidade de um arquivo, arquivos de texto, por exemplo, tem uma randomicidade baixa, já arquivos comprimidos

ou criptografados tem uma entropia mais elevada. Esta técnica aliada ao *Fuzzy hash*, algoritmo capaz de calcular o quanto um arquivo foi modificado através da comparação de *hashes*, construíram a solução de (JOSHI *et al.*, 2021), um *anti-ransomware* que detecta comportamentos de um ataque *ransomware* com um *driver* de mini filtro do *Windows* [1]. Os autores impediram que 95% dos 508 vírus testados causassem qualquer dano à maquina. A taxa de falso positivos foi de 0% na execução de aplicativos como *7-zip*, *Firefox* e *µTorrent*. Segundo os autores, o *anti-ransomware* proposto não foi capaz de identificar os vírus da classe *Jigsaw*, já que estes não se utilizam do mecanismo de leitura e escrita para ferir os arquivos da vítima, outras limitações levantadas por eles é um possível ataque de injeção DLL no *explorer.exe* e também uma adição de mini filtro que poderia desativar os filtros que detectam operações de entrada e saída.

Os *ransomwares* são fortemente dependentes da criptografia, portanto, sem elas, eles não podem existir, pensando nisso o (GENÇ; LENZINI; RYAN, 2018) criou o *UShallNotPass*, uma aplicação que funciona como um guardião das funcionalidades criptográficas do sistema, assim somente aplicações autorizadas podem usar tais funcionalidades. O próprio autor deixa claro que o objetivo do *UShallNotPass* é combater os *ransomwares* mais fortes, deixando de lado os *ransomwares* que possuem chaves pré-configuradas ou que utilizem chamadas http para obter a chave. Com um total de 524 *ransomwares* reais testados a aplicação alcançou a marca de 94% de *ransomwares* detectados. Não foi apresentado no artigo o motivo para 6% dos *ransomwares* não serem detectados, a explicação do autor aponta que estes sejam da classe de *ransomwares* ditos como fracos, entretanto é possível que um *ransomware* possua um algoritmo de geração de chave embutido inviabilizando a defesa desta solução.

Apesar da grande quantidade de soluções em software, eficientes ou não, para combater os *ransomwares*, existem também aqueles que se propõem a proteger os dispositivos de um ataque através de propostas baseadas em hardware. Os autores em (SIDDIQUI; LEE; SAQIB, 2017) formularam um disco rígido com os setores importantes, como o MBR (*Master Boot Record*), responsável pelo *boot* do sistema, protegidos por identificação e autorização. Assim não é possível que um *ransomware* infecte o MBR com código malicioso e sequestre a máquina impossibilitando-a de inicializar. Essa arquitetura garante que nenhum acesso indesejado será feito aos setores protegidos. Caso um acesso a esses setores seja necessário, um dispositivo externo e confiável identifica e autoriza a leitura e escrita nos setores, permitindo, por exemplo, uma atualização do MBR. Esse esquema, proposto pelos autores, protege os setores críticos de uma máquina, mas deixa expostos os setores explorados pelos *Crypto Ransomwares*,

[1] Um *driver* de mini filtro do *Windows* tem a funcionalidade de filtrar operações de entrada e saída do sistema de arquivos, sendo capaz de registrar uma pré-operação para cada chamada de sistema (PORIZEK, 2019).

isto é, os setores no qual estão presentes os arquivos comuns de usuário.

Uma das mais renomadas fabricantes de processadores do mundo, a Intel, também fez sua contribuição no combate aos *ransomwares*. Com uma proposta baseada em *hardware* contendo duas frentes, o *Hardware Shield* e o *Threat Detection Technology* (TDT). O *Hardware Shield* é uma tecnologia a nível de silício que restringe o acesso a áreas de memória pertencentes ao Sistema Básico de Entrada e Saída (*BIOS*), responsável por inicializar a máquina, portanto se algum *malware* tentar acessar estes dados será frustrado pela solução da Intel. Esta técnica é efetiva contra ataques de vírus que se alojam na BIOS para escanear a memória na busca de informações sigilosas, como senhas do usuário e contra *ransomwares* que sequestram a BIOS da máquina. Já o TDT utiliza-se da junção de técnicas de monitoramento de dados provindos do processador e aprendizado de máquina para tentar prever um comportamento malicioso, como o de um *ransomware*. A Intel não divulgou mais detalhes sobre sua implementação, mas afirma que o TDT é efetivo contra ataques de código polimórfico, *fileless malwares*, mineração de criptomoedas e *ransomwares*.

3 METODOLOGIA

É possível ver com mais detalhes a pesquisa efetuada na seção Seção 2, entretanto, nesta seção serão apresentadas as conclusões obtidas com a pesquisa.

3.1 MERCADO DE *RANSOMWARE*

Uma dúvida que pode surgir ao se pensar em *ransomware* é a motivação para o ataque, por que investir tempo para prejudicar a vida alheia? Responder essa pergunta vira uma tarefa simples quando analisa-se alguns dados expostos pela *Unit 42* (UNIT 42, 2021). O maior pagamento por um resgate de *ransomware* em 2020 foi de 10 milhões de dólares, o pagamento foi feito pela Garmin, empresa americana que desenvolve produtos com tecnologia GPS (CYBER SECURITY HUB, 2020). Produzir *ransomware* deixou de ser uma infração praticada por um indivíduo e passou a ser uma das maiores fontes de renda de organizações criminosas. Nesse contexto surge o termo *Ransomware as a Service*, o desenvolvedor produz um *ransomware* e aluga para outras pessoas em troca de um pagamento ou porcentagem nos lucros.

A maioria dos resgates são pagos em criptomoedas dificultando a rastreabilidade até os atacantes, não só isso, mas também a falta de preparo que muitas empresas tem contra ataques cibernéticos acaba facilitando para os criminosos. Nesses últimos dois anos (2020 e 2021) a pandemia intensificou ainda mais os ataques. Segundo uma pesquisa feita pela inteligência de ameaças globais, *Unit 42* (UNIT 42, 2021), os serviços de saúde são os alvos favoritos dos *crackers*, em 2020 os sistemas relacionados a saúde foram os principais alvos de *ransomware*, já que devido ao Covid-19 esses serviços essenciais não poderiam ficar sem funcionar, era de ser esperar que estivessem mais propensos a pagar pelo resgate de dados.

Se defender de um ataque do tipo *ransomware* é uma tarefa difícil que pode ser dividida em 3 etapas, evitar, prevenir e mitigar. Evitar é não acessar *sites* não confiáveis, conscientizar seus funcionários, manter a rede de internet segura e entre outras coisas. Já prevenir é para situações em que a a etapa anterior não foi suficiente e o *ransomware* conseguiu se instalar na máquina ou na rede. Porém, ainda existe a chance do antivírus ou qualquer outro software especializado impedir que o ataque danifique a máquina, por isso a importância de ter um programa que previna ataques deste tipo instalado. Por último, caso não seja possível prevenir que o ataque ocorra, recuperar os arquivos danificados é o último recurso. Muitas vezes *backups* dos arquivos são de fácil acesso e os efeitos do ataque são facilmente revertidos, entretanto é preciso ter cuidado, pois vários *ransomware* tem a capacidade de destruir os *backups* em rede ou local.

3.2 MAIORES *RANSOMWARES* DA HISTÓRIA

Para entender melhor o impacto do *ransomware* na sociedade é preciso estudar o seu desenvolvimento através dos anos. Inicialmente, mais precisamente em 1989, os *ransomwares* eram incipientes, o primeiro deles, AIDS *Trojan* (FLASHPOINT, 2022), utilizava-se de criptografia assimétrica de fácil descriptogração e não obteve muitos ganhos. Com o tempo, os *ransomwares* foram evoluindo e nos dias atuais são muito mais sofisticados contendo até mesmo inteligência artificial. Abaixo estão listados alguns dos maiores *ransomwares* já feitos e suas histórias.

3.2.1 CryptoLocker

Em 2013, considerado o primeiro vírus a se espalhar através de *botnet*, *CryptoLocker* começou a infectar computadores *Windows* por diversos países. O *malware* usava uma chave de criptografia assimétrica RSA de 1024 *bits* para criptografar os arquivos das vítimas e mantivera as chaves privadas no servidor de controle. Após criptografar os arquivos, o *ransomware* apresentava uma tela ordenando um resgate em *bitcoin* ou vouncher pré-pago, além de uma data limite para o pagamento. Não havia garantia de que após o pagamento os arquivos seriam descriptografados. O *CryptoLocker* era facilmente removível do computador, mas os arquivos ainda continuavam criptografados, somente um ano depois dos ataques quando a Operação Tovar, criada para acabar com a *botnet* usada pelo *CryptoLocker*, obteve acesso ao banco de dados que continham as chaves privadas das vítimas é que foi possível descriptografar uma grande parte dos arquivos de lesados pelo ataque. Estima-se que o ataque arrecadou três milhões de dólares para os criminosos (EXCELLENCE, 2022).

3.2.2 CryptoWall

Considerado o sucessor do *CryptoLocker* pela semelhança de seus ataques, o *CryptoWall* utilizava-se de uma chave de criptografia assimétrica RSA de 2048 *bits* para criptografar o arquivos da vítima. Projetado para infectar todas as versões do *Windows* o *malware* tinha uma tela de resgate semelhante ao do seu antecessor e ordenava um resgate de 500 dólares, pagos exclusivamente em *bitcoin*. Com surgimento em 2013, se tornou famoso em 2014, sendo considerado a ameaça número um de ataques do tipo *ransomware* (FLASHPOINT, 2022). O vírus se espalhava pelo *spam* de *emails* e um relatório feito pela Cyber Threat Alliance (CTA) sugere que o *malware* tenha arrecadado 325 milhões de dólares (THE HACKER NEWS, 2015).

3.2.3 Petya

Petya é um *ransomware* sofisticado que ataca os sistemas *Windows* e opera de duas maneiras. Primeiramente o vírus tenta conseguir privilégios de administrador

através de um *script* carregado em um PDF enviado para o *email* da vítima. Caso obtenha sucesso o *malware* instala-se no Registro Mestre de Inicialização (MBR) e impede que o computador inicialize. Depois, em caso de não conseguir privilégios de administrador o programa malicioso criptografa os arquivos do usuário (AVAST, 2016). *Petya* surgiu em 2016 e não foi tão bem sucedido quanto o *CryptoWall*, mas seu *modus operand* inspirou a criação de diversos outros *ransomwares*, incluindo o *notPetya*, um dos *ransomwares* que causaram mais danos em dinheiro na história dos ataques cibernéticos.

3.2.4 notPetya

Sucessor do *Petya*, *notPetya* conseguiu alcançar uma quantidade muito maior de computadores que seu antecessor e acabou se tornando conhecido e temido. Estima-se que *notPetya* tenha causado 10 bilhões de dólares em danos. O motivo para esse sucesso se dá pelo fato do vírus não mais depender de *spam* de *email* para propagar-se. Uma vulnerabilidade na implementação do protocolo SMB da *Microsoft*, conhecida como *EternalBlue*, possibilitou que os *crackers* executassem códigos arbitrários no computador das vítimas, fazendo com que o vírus se espalhasse com muito mais facilidade (THE ECONOMIC TIMES, 2022). Apesar dos altos danos em dinheiro, esse vírus tinha um resgate consideravelmente baixo em comparação com outros *ransomwares* da época. Este fato, alinhado ao fato de que *notPetya* foi projetado de uma forma para que os dados da vítima nunca pudessem ser recuperados, mesmo sob pagamento, levou os Estados Unidos, Reino Unido e Austrália a culpar a Rússia pelo ataque (FLASHPOINT, 2022).

3.2.5 WannaCry

A falha *EternalBlue* não foi usada somente pelo *notPetya*, o *WannaCry* se espalhou por mais de 230 mil computadores em 150 países usando a vulnerabilidade no sistema da *Microsoft* (FLASHPOINT, 2022). Após infectar uma máquina o vírus tinha uma capacidade de propagação muito alta devido suas características de *worm*, *malware* que se propaga pela rede. Além de criptografar os dados da vítima, serviços do sistema como o de recuperação de dados eram desabilitados. O resgate dos arquivos era de 300 dólares, pagos em *bitcoin*, e deveriam ser pagos em até três dias, caso contrário todos os dados criptografados eram destruídos. A Casa Branca culpa a Coreia do Norte pelo ataque e pelos quatro bilhões de dólares que o *ransomware* causou em danos (KASPERSKY, 2022).

3.3 PROPOSTAS E JUSTIFICATIVAS

Detectar que um computador está sob ataque de um *ransomware* é complicado e, até o momento, não existe uma solução que detecte com certeza a presença de *ransomware* sem gerar falsos positivos. As técnicas usadas na detecção variam entre as diversas subáreas da computação, como análise de poder computacional, aprendizado de máquina, redes neurais e até soluções em hardware como as da Intel (INTEL, 2021). Escolher em que área focar para desenvolver uma solução computacional capaz de detectar ataques do tipo *ransomware* não é uma tarefa fácil.

Primeiramente, notou-se que entres as melhores soluções se destacam as que não utilizam inteligencia artificial, provavelmente por causa da dificuldade encontrada para montar um ambiente de teste disposto a prover treinamento suficiente para que o software tenha bons resultados em um cenário real, além da dificuldade de encontrar o conjunto ideal de treino que não deixe a aplicação super treinada, isto é, incapaz de operar fora do ambiente de teste. Por fim, idealizou-se uma solução em módulo do *kernel Linux* capaz de prevenir que um ataque ocorra.

A solução pensada tem como foco expor um ataque do tipo *ransomware* em um ambiente Linux através da verificação de disparidade na modificação de arquivos. O módulo em *kernel* será capaz de determinar quando ele deve agir e impedir que uma modificação a um arquivo seja efetuada. Uma prova de conceito foi desenvolvida, após um extenso estudo a respeito das tecnologias mais adequadas para resolução do caso. Mais detalhes da implementação e motivos da escolha de determinada tecnologia estão na Seção 4.

É importante ressaltar que a arquitetura proposta tem como objetivo a detecção e prevenção de ataques de *malwares* da classe *Crypto Ransomware*. Como explicado anteriormente, essa classe de vírus tem como alvo criptografar os arquivos da vítima e impedir o acesso a eles, entretanto o uso desta solução isolada não contribui para com ataques que visam, por exemplo, sequestrar o MBR da máquina, entre outros ataques que não só bloqueiam ou restringem o acesso ao sistema operacional mas também podem impedir que a máquina inicialize.

4 DESENVOLVIMENTO DA PROVA DE CONCEITO

Nessa seção será apresentado o motivo da escolha de certas tecnologias para o desenvolvimento do trabalho e os passos que foram necessários para chegar em uma prova de conceito executável. Após a confecção da prova de conceito foi desenvolvido também um módulo de *kernel Linux* com os aprendizados obtidos durante a etapa anterior. O desenvolvimento do *LKM* (*Loadable Kernel Module*) pode ser visto na Seção 5.1.

4.1 ESCOLHAS DE TECNOLOGIAS

Como o problema que precisa ser solucionado tem aspectos que indicam a necessidade de um controle em mais baixo nível dos acontecimentos, incluindo gerenciamento de recursos e acesso a módulos do *kernel* do *Linux*, a linguagem C acaba por ser a mais adequada. Para confeccionar a prova de conceito foi pensado inicialmente no uso do *inotify*, uma API que monitora chamadas ao sistema de arquivos. Com ela foi possível construir uma aplicação que emite um alerta quando um arquivo é acessado, entretanto, não mais que isso, é possível com essa API. O *inotify* não suporta controle de acesso a arquivos e também não é possível ler e modificar arquivos antes que sejam acessados pela aplicação chamadora. Uma alternativa para o *inotify* foi encontrada, o *fanotify*. Portar o código do *inotify* para o *fanotify* foi uma tarefa fácil. O *fanotify* é mais completo e possui todas as características de uma API de acesso ao sistema de arquivos. Com o *fanotify* foi possível construir uma prova de conceito capaz de, além de detectar acesso a arquivos, já implementado anteriormente, permitir ou negar modificações em arquivos baseado na porcentagem de *bytes* modificados.

4.2 PROVA DE CONCEITO

A prova de conceito apresentada tem como objetivo estabelecer que é possível construir uma aplicação avançada capaz de impedir ataques *ransomware* através da análise computacional. A *PoC (Proof of Concept)* é um programa *daemon*, isto é, executa como um processo em segundo plano, feito para *Linux* e roda somente com privilégios de administrador.

Após a *Poc* estar executando, arquivos dentro da partição montada e especificada, quando acessados, lançam um evento de permissão que será recebido pelo módulo do *kernel fanotify*. Antes de permitir que o usuário acesse o arquivo, um *backup* dele é criado para fins de comparação com as mudanças feitas pelo usuário. Quando o arquivo deixar de ser usado e for fechado, um evento do tipo *FAN_CLOSE_WRITE* é gerado e enviado ao *kernel*. Neste momento, antes de fechar o arquivo, a *PoC* compara a porcentagem de *bytes* que mudaram desde o arquivo original. Caso as mudanças

ultrapassem uma determinada porcentagem de *bytes*, o programa substitui o arquivo modificado pelo *backup* e deleta-o. Para a *PoC* foi escolhido arbitrariamente uma taxa de mudanças permitidas de no máximo 50% dos *bytes* do arquivo. Este fluxo pode ser observado na Figura 1.

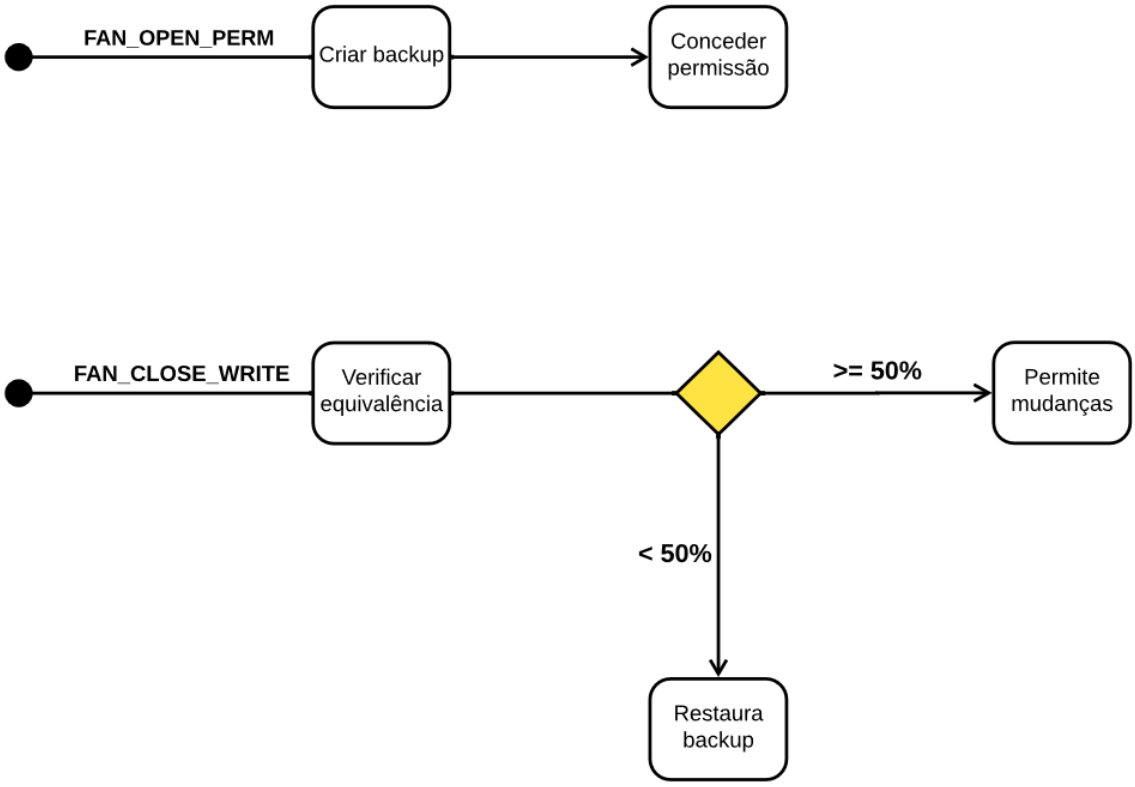

Figura 1 – Diagrama exemplificando o funcionamento da *PoC*.

4.2.1 Manipulador de eventos

Para que seja possível manipular os eventos de abertura e fechamento de arquivo foi implementado uma função responsável por interceptar qualquer tentativa de abrir um arquivo no diretório montado alvo. A implementação usa o *fanotify* e pode ser vista no Código 1. Caso o evento seja do tipo *FAN_OPEN_PERM*, isto é, um pedido do usuário para acessar o arquivo, um *backup* do arquivo é criado e a permissão é concedida. Se o evento for do tipo *FAN_CLOSE_WRITE*, isto é, um fechamento de arquivo com alterações, uma verificação de equivalência com o arquivo original é feita, veja Seção 4.2.2, caso o arquivo possua mais de 50% dos *bytes* alterados a modificação não surge efeito no arquivo e o *backup* é restaurado.

```
1 static void handle_events(int fd) {
2  // inicialização de variáveis
3  for (;;) {
4   len = read(fd, buf, sizeof(buf));
5   metadata = buf;
```

```
6
7   while (FAN_EVENT_OK(metadata, len)) {
8    if (metadata->fd >= 0) {
9     if (metadata->pid != getpid()) {
10     if (metadata->mask & FAN_OPEN_PERM) {
11      snprintf(procfd_path, sizeof(procfd_path),
12               "/proc/self/fd/%d", metadata->fd);
13
14      path_len = readlink(procfd_path, path, sizeof(path) - 1);
15      path[path_len] = '\0';
16
17      filename = strrchr(path, '/');
18      sprintf(bkpath, "/home/user/%s.swp", ++filename);
19
20      copyFile(metadata->fd, bkpath);
21
22      response.fd = metadata->fd;
23      response.response = FAN_ALLOW;
24      write(fd, &response, sizeof(response));
25     } else {
26      if (metadata->mask & FAN_CLOSE_WRITE) {
27       snprintf(procfd_path, sizeof(procfd_path),
28                "/proc/self/fd/%d", metadata->fd);
29
30       path_len = readlink(procfd_path, path, sizeof(path) - 1);
31       path[path_len] = '\0';
32
33       filename = strrchr(path, '/');
34       sprintf(bkpath, "/home/user/%s.swp", ++filename);
35
36       file = fdopen(metadata->fd, "rb");
37       fseek(file, 0, SEEK_END);
38
39       filelen = ftell(file);
40       filebytes = (char *) malloc(filelen * sizeof(char));
41       filesize = setByteArray(file, filebytes, filelen);
42
43       file = fopen(bkpath, "rb");
44       fseek(file, 0, SEEK_END);
45       filelen = ftell(file);
46
47       bkfilebytes = (char *) malloc(filelen * sizeof(char));
48       setByteArray(file, bkfilebytes, filelen);
49
50       float compare = compareByteSimilarity(filebytes, filesize,
51                                             bkfilebytes,
52                                             filelen * sizeof(char));
53
```

```
54          if (compare >= 50.0f) {
55           remove(bkpath);
56          } else {
57           remove(path);
58           rename(bkpath, path);
59          }
60          free(filebytes);
61          free(bkfilebytes);
62         }
63        }
64       }
65       close(metadata->fd);
66      } else {
67       // OVERFLOW
68      }
69      metadata = FAN_EVENT_NEXT(metadata, len);
70     }
71    }
72 }
```

Código 1 – Manipulador de eventos de arquivo

4.2.2 Cálculo de equivalência de *bytes*

O cálculo para determinar se um arquivo pode estar sendo criptografado é feito pela função do Código 2. Os *bytes* do arquivo original são comparados um a um com os *bytes* do arquivo modificado a fim de calcular a porcentagem de *bytes* modificados. A função retorna a porcentagem de *bytes* equivalentes. Este algoritmo foi utilizado somente na *PoC*, para uma abordagem mais técnica veja Seção 5.1.2.

```
1 float compareByteSimilarity(const char *a, unsigned long asize, const char
     *b, unsigned  long bsize) {
2     unsigned long min = MIN(asize, bsize);
3     unsigned long max = MAX(asize, bsize);
4     unsigned long unequalCount = max - min;
5     for (int i = 0; i < min; i++)
6         if (a[i] != b[i])
7             unequalCount++;
8     return 100.0f - (float) unequalCount * 100.0f / (float) max;
9 }
```

Código 2 – Calcula porcentagem de equivalência entre vetores de *bytes*

4.2.3 Teste

Para testar a solução criada foi desenvolvido um programa de teste. O programa de teste é responsável por abrir um arquivo e modificá-lo para que haja diferenças

do arquivo original. O grau de mudanças ainda não é algo automatizado e pode ser mudado pelo código do programa de teste.

5 DESENVOLVIMENTO DO SISTEMA

Com a finalização da *PoC* ficou nítido que seria possível avançar para um desenvolvimento mais robusto do sistema, já que o objetivo da *PoC* foi alcançado. Para essa etapa do desenvolvimento pensou-se em trazer a solução de defesa para dentro do núcleo do próprio sistema operacional, obtendo mais segurança e tornando-se imperceptível ao usuário. Estender o núcleo do *Linux* é possível através do módulo carregável do núcleo (*LKM*), um arquivo objeto que pode ser carregado junto ao núcleo base em execução.

5.1 DESENVOLVIMENTO DO MÓDULO DE *KERNEL*

O *kernel* é o componente principal do sistema operacional *Linux*. É a principal interface entre o *hardware* de um computador e seus elementos lógicos. Responsável por administrar os recursos, entradas e saídas de dados de cada processo. No *Linux* é possível integrar módulos ao *kernel* em execução incrementando as funcionalidades já existentes e até mesmo criando novas funcionalidades. Neste trabalho três chamadas de sistema foram sobrescritas para incluir os efeitos requeridos pelo fluxo proposto na Figura 2. Como referência para produção do trabalho utilizou-se não só da documentação oficial do *Kernel* (LINUX MAN-PAGES, 2022), mas também do código fonte da versão 5.15.0 do *kernel* (BOOTLIN, 2022).

5.1.1 Manipulando eventos

No *kernel* do *Linux* são definidas todas as chamadas essenciais do sistema, como as de leitura e escrita de arquivos. Os endereços de todas essas funções do sistema operacional ficam armazenados em uma tabela estática em uma área de memória protegida dentro do *kernel*. Na versões recentes do *kernel* o endereço dessa tabela não é exposto à outros níveis do sistema, mas é possível encontrá-la com busca por força bruta na memória do *kernel*. Com acesso a tabela é possível modificar os endereços de cada entrada, assim, pode-se modificar as chamadas de sistema apontando-as para endereços distintos do original. Graças a essa liberdade, foi montado uma esquema capaz de controlar os acessos aos arquivos e identificar uma tentativa de criptografar os arquivos do usuário. O esquema pode ser visto na Figura 2.

Quando a máquina, executando o *kernel* modificado, recebe uma chamada de sistema solicitando a abertura de um arquivo, seja de leitura ou escrita, ocorre uma filtragem dos arquivos que devem ser analisados e quais devem ser ignorados. Os parâmetros de filtros podem ser facilmente mudados. Neste projeto o diretório *home* do *Linux* é o escolhido, por ser o diretório que contém a maioria dos arquivos importantes do usuário.

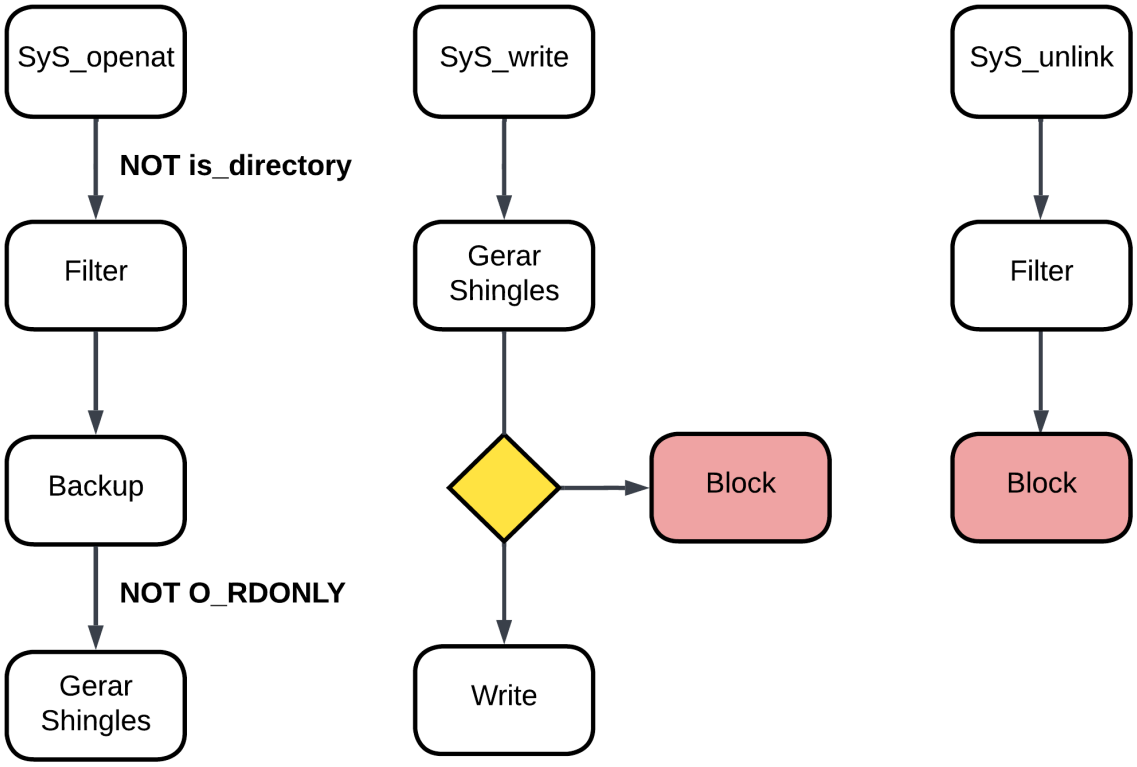

Figura 2 – Diagrama da manipulação de chamadas do sistema.

Após a filtragem, um *backup* do arquivo é criado para impedir que um atacante leia os *bytes* de um arquivo em somente leitura e crie uma cópia criptografada dele, excluindo o original. Se a chamada de sistema for de somente leitura o fluxo encerra-se nesta etapa.

Caso a abertura do arquivo implique em uma escrita, o atacante pode sobrescrever os *bytes* originais do arquivo pelos criptografados. Para evitar essa forma de ataque é feito um *hash* baseado na técnica de *hash* sensível à localidade do arquivo aberto e após a chamada de escrita ao sistema operacional os *hashes* do arquivo original são comparados ao modificado para validar se essa é uma escrita incremental ou se há uma tentativa de sobrescrever os *bytes* pelos de um criptografado. Para entender melhor como é feito o *hash* do arquivo veja Seção 5.1.2. Na condição de que seja identificado um ataque a escrita é bloqueada.

Como já citado anteriormente, um *backup* do arquivo é criado sempre que há uma leitura, para garantir que não haja chances de ataque que envolvam deletar o arquivo original e criar uma cópia criptografada. Mas é importante ressaltar que dessa forma os arquivos de *backup* ficariam desprotegidos caso um *ransomware* tentasse apagá-los. Para evitar esse problema houve uma sobrescrita da chamada de sistema *unlink* para bloquear qualquer tentativa de excluir as cópias de segurança.

```
1 static asmlinkage int hooked_openat(const struct pt_regs *regs) {
2     // inicialização de variáveis
```

```
3     hash = createHashPath(path);
4     hashPath[40] = '\0';
5     sprintf(hashPath, "/backup/%s", hash);
6
7     file = filp_open(path, O_RDONLY | O_LARGEFILE, 0777);
8     size = vfs_llseek(file, 0, SEEK_END);
9     vfs_llseek(file, 0, SEEK_SET);
10    file->f_pos = 0;
11
12    if (size < 3 || size > 524288000) {
13        filp_close(file, NULL);
14        goto free2;
15    }
16
17    buffer = kmalloc(size + 1, GFP_KERNEL);
18    ret = kernel_read(file, buffer, size, &file->f_pos);
19    buffer[ret] = '\0';
20    filp_close(file, NULL);
21    file = filp_open(hashPath, O_CREAT | O_WRONLY | O_TRUNC, 0777);
22    file->f_pos = 0;
23    size = kernel_write(file, buffer, ret, &file->f_pos);
24    filp_close(file, NULL);
25
26    if ((regs->dx&O_ACCMODE) == O_RDONLY)
27        goto free3;
28
29    pivotIndex = ret / 2;
30
31    pivot[0] = buffer[pivotIndex];
32    pivot[1] = buffer[pivotIndex + 1];
33    pivot[2] = '\0';
34
35    totalShingles = generateShingles(buffer, ret, pivot, &shingles, &
      shinglesSizes);
36
37    if (!createHashes(pivot, &shingles, &shinglesSizes, totalShingles, &
      write, &writeSize))
38        goto free;
39
40    saveFile(hash, write, writeSize);
41    // liberar memória
42    return (*original_openat)(regs);
43 }
```

Código 3 – *SyS_openat syscall* modificada

```
1 static asmlinkage int hooked_write(const struct pt_regs *regs) {
2     // inicialização de variáveis
```

```
3     file = fget(fd);
4     size = vfs_llseek(file, 0, SEEK_END);
5     vfs_llseek(file, 0, SEEK_SET);
6     file->f_pos = 0;
7     fput(file);
8
9     if (size > 0)
10        goto exit;
11
12    result = kmalloc(33, GFP_KERNEL);
13
14    if (!md5(path, strlen(path), &result))
15        goto free2;
16
17    hashPath[37] = '\0';
18    sprintf(hashPath, "/tmp/%s", result);
19
20    file = filp_open(hashPath, O_RDONLY, 0777);
21
22    if (IS_ERR(file))
23        goto free2;
24
25    size = vfs_llseek(file, 0, SEEK_END);
26    vfs_llseek(file, 0, SEEK_SET);
27    file->f_pos = 0;
28
29    hashTxt = kmalloc(size + 1, GFP_KERNEL);
30    ret = kernel_read(file, hashTxt, size, &file->f_pos);
31    hashTxt[ret] = '\0';
32
33    pivot[0] = hashTxt[0];
34    pivot[1] = hashTxt[1];
35    pivot[2] = '\0';
36
37    totalShingles = generateShingles((char *)regs->si, regs->dx, pivot, &
      shingles, &shinglesSizes);
38
39    if (!createHashes(pivot, &shingles, &shinglesSizes, totalShingles, &
      write, &writeSize))
40        goto free1;
41
42    if (!compareHashes(hashTxt, write))
43        goto block;
44    // liberar memória
45    return (*original_write)(regs);
46
47 block:
```

```
48     return (-EACCES);
49 }
```

Código 4 – *SyS_write syscall* modificada

5.1.2 Similaridade de *bytes*

Hash sensível à localidade (*LSH*) é uma técnica capaz de produzir *hashes* iguais para entradas de dados semelhantes, a maior aplicabilidade dessa técnica é na procura de vizinhos mais próximos em grandes dimensões (DASGUPTA; KUMAR; SARLOS, 2011), mas o princípio de seu funcionamento também é aplicável em outras áreas, como na detecção de *spams*. Seguindo essa linha, foi desenvolvido um algoritmo para geração de *hashes* dos arquivos, afim de comparar duas versões do mesmo arquivo e determinar se ele foi criptografado.

Para simplificar a explicação do algoritmo proposto, notação hexadecimal será utilizado no lugar dos bytes do arquivo, mas é importante ressaltar que o algoritmo executado tem como entrada os bytes de um arquivo qualquer. Na Figura 3 pode-se ver o funcionamento do algoritmo de geração de *hash*.

Primeiramente os bytes pivôs são selecionados, no caso da Figura 3, **CB** foram escolhidos. Logo então, os bytes são separados em conjuntos delimitados pelo pivô e cada grupo tem seu *hash* gerado, o *hash* de cada conjunto é chamado de *shingle*. Por último, concatena-se todos os *shingles* para construir o *hash* final.

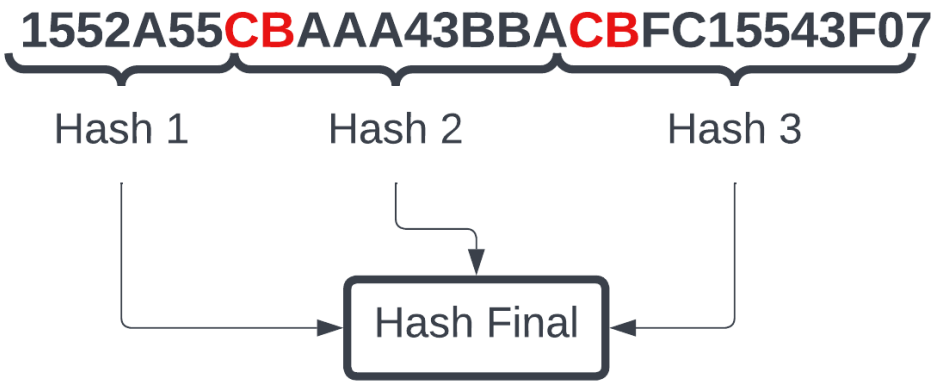

Figura 3 – Geração de *Shingles*.

Com um *hash* gerado dessa forma, se o arquivo for alterado, somente parte do *hash* mudará, assim, é possível identificar uma alteração drástica em um arquivo caso os *hashes* sejam completamente diferentes. Por exemplo, mesmo que um arquivo PDF tenha seu texto completamente modificado, essa alteração ainda será válida, já que o arquivo PDF contém toda uma estrutura interna que permanecerá semelhante. É possível ver o código da geração de *shingles* no Código 5.

```
unsigned short generateShingles(char *buffer, const size_t bufferSize,
    const char *pivot, char ***shingles, unsigned short **shinglesSizes) {
    size_t i;
    unsigned short shingleIndex, counter;

    shingleIndex = 1;
    counter = 0;
    *shingles = kmalloc(sizeof(char *), GFP_KERNEL);
    *shinglesSizes = kmalloc(sizeof(unsigned short), GFP_KERNEL);
    (*shingles)[0] = buffer;

    for (i = 2; i < bufferSize - 1; i++) {
        if (shingleIndex > 49)
            break;

        if (buffer[i] == pivot[0] && buffer[i + 1] == pivot[1]) {
            *shinglesSizes = krealloc(*shinglesSizes, shingleIndex * sizeof
    (unsigned short), GFP_KERNEL);
            (*shinglesSizes)[shingleIndex - 1] = i - counter;
            *shingles = krealloc(*shingles, (shingleIndex + 1) * sizeof(
    char *), GFP_KERNEL);
            (*shingles)[shingleIndex++] = &buffer[i];

            counter = i++;
        }
    }

    *shinglesSizes = krealloc(*shinglesSizes, shingleIndex * sizeof(
    unsigned short), GFP_KERNEL);
    (*shinglesSizes)[shingleIndex - 1] = bufferSize - counter;

    return shingleIndex;
}
```

Código 5 – Código para geração de *Shingles*

5.1.3 Testes

Todo desenvolvimento do módulo e os testes foram feitos em réplicas da mesma máquina virtual com sistema operacional *Ubuntu 22.04* com *Kernel* versão *5.15.0*.

Para testar as funcionalidades acrescentadas no *kernel* do *Linux* foram confeccionados dois *scripts* em *Python*. O primeiro atua como um *Crypto Locker* criptografando os arquivos no diretório *home* do usuário. O código pode ser visto no Código 6. O objetivo desse teste é validar que é possível identificar uma tentativa de criptografar os arquivos do usuário e impedir danos à vítima do ataque *ransomware*. Já o segundo *script* faz leituras e escritas não maliciosas em ativos do usuário, afim de provar que

operações legítimas não foram impactadas. Veja o *script* em Código 7.

No primeiro teste concluiu-se que os objetivos de proteger os arquivos do usuário contra ataques de *ransomware* que criptografam arquivos da vítima foi alcançado. Já no segundo teste, o objetivo de ser imperceptível ao usuário foi parcialmente alcançado. O módulo do *kernel* trabalha de forma silenciosa não impactando as tarefas do usuário na maior parte do tempo, mas alguns usuários gostam de criptografar seus arquivos e o *kernel* irá bloquear a operação. Outro fator impactante para o usuário, é o fato do arquivo de texto plano possuir uma equivalência direta entre o texto escrito e os bytes do arquivo, fazendo com que a sobrescrita completa de um arquivo de texto simples gere um alarme falso. Para substituir por completo um arquivo de texto é necessário excluí-lo e criá-lo novamente.

Também foram elaborados dois *scripts* para medir o impacto do módulo na performance do sistema. Um deles faz mil leituras consecutivas em um arquivo e calcula o tempo médio de execução. O outro *script* segue o mesmo padrão, mas desta vez com mil escritas em um arquivo. Pode-se ver o resultado na Tabela 1.

Modo	Sem módulo	Com módulo
Leitura	0.41	1.26
Escrita	1.96	4.38

Tabela 1 – Comparação do tempo de execução em milissegundos

```
def iterateDir(baseDir):
    for d in os.scandir(baseDir):
        if d.is_file():
            yield d
        else:
            yield from iterateDir(d.path)

def encrypt(path):
    extension = path.suffix.lower()

    path = str(path)

    with open(path, 'rb+') as f:
        data = f.read()

        f.seek(0)

        data = bytes(data)

        key = RSA.importKey(pub64)
        sessionKey = os.urandom(16)
        cipher = PKCS1_OAEP.new(key)
        encryptedSessionKey = cipher.encrypt(sessionKey)
```

```
24          cipher = AES.new(sessionKey, AES.MODE_EAX)
25          ciphertext, tag = cipher.encrypt_and_digest(data)
26
27          [ f.write(x) for x in (encryptedSessionKey, cipher.nonce, tag,
        ciphertext) ]
28          f.truncate()
29
30  baseDir = "/home"
31
32  for p in iterateDir(baseDir):
33      path = Path(p)
34      encrypt(path)
```

Código 6 – *Script* para criptografar arquivos a partir de um diretório base

```
with open("txt_to_append.txt", 'a') as f:
    f.write(" Teste feito.")

imgPath = "img_to_flip.png"
image = Image.open(imgPath)
flipped = image.transpose(Image.FLIP_LEFT_RIGHT)
flipped.save(imgPath)

output = PdfFileWriter()
with open("pdf_to_addpage.pdf", 'rb+') as f:
    pdf = PdfFileReader(f)
    f.seek(0)
    output.addPage(pdf.getPage(0))
    output.addPage(pdf.getPage(0))
    output.write(f)
    f.truncate()
```

Código 7 – *Script* que executa ações legítimas de usuário

5.2 CONSIDERAÇÕES FINAIS

Neste trabalho foi planejado pesquisar a respeito das técnicas de detecção e proteção contra *ransomwares*, para então desenvolver uma solução computacional que atingisse o objetivo de detectar e proteger contra essa ameaça. Os *ransomwares* são programas maliciosos que tornam inacessíveis os dados armazenados em um computador, usando criptografia, e que exige pagamento de resgate. Nos tempos atuais é inevitável que, com o adendo da tecnologia, o ser humano fique dependente dos caprichos da tecnologia. Muitos trabalhos são feitos com o auxílio computacional e alguns até mesmo reféns da tecnologia. Essa dependência gera perigos que não podem ser ignorados. Um deles é o *ransomware*, capaz de bloquear o acesso à informações que muitas vezes são de extremo valor para seu dono, prejudicando-o de várias maneiras. Como já citado neste trabalho diversas empresas tem gastos milionários com essa classe de *malware*, inclusive hospitais que se submetem ao pagamento exigido pelos criminosos para não colocar vidas em perigo.

O objetivo geral deste trabalho foi propor um software capaz de detectar e prevenir *ransomwares* em ambientes *Linux*. A partir disso, foi possível desenvolver um módulo do *kernel* do *Linux* como uma solução proposta a resolver este problema. O módulo cumpre o objetivo de impedir a criptografia de ativos do usuário, mas tem suas ressalvas quando tratando-se de passar despercebido pelo usuário da máquina. Apesar das limitações da solução, este estudo é apenas o início das soluções de detecção de *ransomwares* que partam diretamente das funções do sistema operacional, ainda há muito que ser estudado e melhorado nessa área. Muitas soluções nessa área podem futuramente já virem incluídas no próprio sistema operacional, sem a necessidade

de carregar um módulo para esta camada de defesa.

Como pesquisa futura pode-se focar na distinção de ataques para operações legítimas do usuário. É de grande importância na segurança que a proteção de ativos seja estabelecida sem deteriorar a usabilidade do usuário, por isso essa seria uma grande melhoria nessa linha de pesquisa. Outro ponto de extrema relevância é na detecção de ataques, ser possível identificar um ataque através de heurísticas seria uma melhoria muito bem-vinda.

REFERÊNCIAS

ASLAN, Ömer Aslan; SAMET, Refik. **A Comprehensive Review on Malware Detection Approaches**. [*S.l.*], 2020. P. 6249–6271. DOI: 10.1109/ACCESS.2019.2963724. Citado na p. 8.

AVAST. **Inside Petya and Mischa ransomware**. [*S.l.*], jul. 2016. Disponível em: <https://blog.avast.com/inside-petya-and-mischa-ransomware>. Acesso em: 12 ago. 2022. Citado na p. 17.

BLEEPINGCOMPUTER. **Adobe Flash Player is officially dead tomorrow**. [*S.l.*], dez. 2020. Disponível em: <https://www.bleepingcomputer.com/news/security/adobe-flash-player-is-officially-dead-tomorrow/>. Acesso em: 6 jul. 2022. Citado na p. 9.

BOOTLIN. **Bootlin – Embedded Linux and kernel engineering**. [*S.l.*]. Disponível em: <https://elixir.bootlin.com/linux/v5.15/source>. Acesso em: 31 ago. 2022. Citado na p. 24.

CANALTECH. **85% das empresas sofrem ao menos um ataque de ransomware por ano, diz estudo**. [*S.l.*], mar. 2022. Disponível em: <https://canaltech.com.br/seguranca/85-das-empresas-sofrem-pelo-menos-1-ataque-virtual-por-ano-aponta-estudo-210542/>. Acesso em: 2 mar. 2022. Citado na p. 8.

CFO DIVE. **Cyber extortion surges 78% on spread of 'ransomware-as-a-service'**. [*S.l.*], mai. 2022. Disponível em: <https://www.cfodive.com/news/cyber-extortion-surges-78-spread-ransomware-as-a-service/621133/>. Acesso em: 3 mai. 2022. Citado na p. 8.

CONSTANTINESCU, Cornel; SESHADRI, Sangeetha. **Sentinel: Ransomware Detection in File Storage**. *In:* PROCEEDINGS of the 14th ACM International Conference on Systems and Storage. Haifa, Israel: Association for Computing Machinery, 2021. (SYSTOR '21). DOI: 10.1145/3456727.3463834. Disponível em: <https://doi.org/10.1145/3456727.3463834>. Citado na p. 12.

CYBER SECURITY HUB. **Garmin Pays $10 Million To Ransomware Hackers Who Rendered Systems Useless**. [*S.l.*], ago. 2020. Disponível em: <https://www.cshub.com/attacks/articles/incident-of-the-week-garmin-pays-

10-million-to-ransomware-hackers-who-rendered-systems-useless>. Acesso em: 19 jan. 2022. Citado na p. 15.

DASGUPTA, Anirban; KUMAR, Ravi; SARLOS, Tamas. Fast Locality-Sensitive Hashing. *In:* PROCEEDINGS of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. San Diego, California, USA: Association for Computing Machinery, 2011. (KDD '11), p. 1073–1081. DOI: 10.1145/2020408.2020578. Disponível em: <https://doi.org/10.1145/2020408.2020578>. Citado na p. 28.

EXCELLENCE. **Cryptolocker Ransomware Attack: A Brief History Explained!** [*S.l.*], 2022. Disponível em: <https://excellence-it.co.uk/insights/cryptolocker-ransomware-attack-explained/>. Acesso em: 12 ago. 2022. Citado na p. 16.

FLASHPOINT. **The History and Evolution of Ransomware Attacks**. [*S.l.*], jul. 2022. Disponível em: <https://flashpoint.io/blog/the-history-and-evolution-of-ransomware-attacks/>. Acesso em: 11 ago. 2022. Citado nas pp. 16, 17.

GENÇ, Ziya Alper; LENZINI, Gabriele; RYAN, Peter Y. A. No Random, No Ransom: A Key to Stop Cryptographic Ransomware. *In:* GIUFFRIDA, Cristiano; BARDIN, Sébastien; BLANC, Gregory (Ed.). **Detection of Intrusions and Malware, and Vulnerability Assessment**. Cham: Springer International Publishing, 2018. P. 234–255. Citado na p. 13.

INTEL. **1th Gen Intel Core vPro Mobile Platform PCs Feature the Industry's Only Silicon-Enabled Threat Detection**. [*S.l.*], 2021. Citado na p. 18.

JOSHI, Yash Shashikant *et al.* **Signature-less ransomware detection and mitigation**. [*S.l.*], mai. 2021. Citado nas pp. 10, 13.

KASPERSKY. **O que é o ransomware WannaCry?** [*S.l.*]. Disponível em: <https://www.kaspersky.com.br/resource-center/threats/ransomware-wannacry>. Acesso em: 12 ago. 2022. Citado na p. 17.

LINUX MAN-PAGES. **The Linux man-pages project**. [*S.l.*]. Disponível em: <https://www.kernel.org/doc/man-pages>. Acesso em: 31 ago. 2022. Citado na p. 24.

LISKA, Allan; GALLO, Timothy. **Ransomware: Defendendo-se da Extorsão Digital**. [*S.l.*], 2017. P. 19–32. Citado na p. 9.

LU, Yan; GRAHAM, Jonathan; LI, Jiang. Deep learning based malware classification using deep residual network. *In:* PROCEEDINGS of the 2019 Modeling, Simulation and Visualization Student Capstone Conference. [*S.l.*: *s.n.*], 2019. P. 1–6. Citado na p. 12.

MELTON, Jacob D. **DETECTING RANSOMWARE THROUGH POWER ANALYSIS**. Monterey, CA, jun. 2018. Citado na p. 12.

NETAPP. **Ransomware Types: 5 Common Types of Ransomware Attack Top Security Tips for IT Teams**. [*S.l.*], mai. 2022. Disponível em: <https://cloud.netapp.com/blog/rps-blg-5-common-types-of-ransomware-attack-top-security-tips-for-it-teams>. Acesso em: 21 jun. 2022. Citado na p. 9.

PANDEY, Poonam; PRABHAKAR, Radhika. An analysis of machine learning techniques (J48 & AdaBoost)-for classification. *In:* 2016 1st India International Conference on Information Processing (IICIP). [*S.l.*: *s.n.*], 2016. P. 1–6. DOI: 10.1109/IICIP.2016.7975394. Citado na p. 12.

PORIZEK, David. **Transparent Encryption with Windows Minifilter Driver**. [*S.l.*], 2019. Citado na p. 13.

POUDYAL, Subash; DASGUPTA, Dipankar. **AI-Powered Ransomware Detection Framework**. Memphis, TN, dez. 2020. Citado na p. 12.

SC MAGAZINE. **Linux malware gaining favor among cybercriminals**. [*S.l.*], jun. 2017. Disponível em: <https://www.scmagazine.com/news/iot/linux-malware-gaining-favor-among-cybercriminals>. Acesso em: 1 jul. 2021. Citado na p. 10.

SIDDIQUI, Ali Shuja; LEE, Chia-Che; SAQIB, Fareena. **Hardware based protection against malwares by PUF based access control mechanism**. [*S.l.*], 2017. P. 1312–1315. DOI: 10.1109/MWSCAS.2017.8053172. Citado na p. 13.

TAFKOV, Stefan; MINCHEV, Zlatogor. **Ransomware Detection and Neutralization System**. [*S.l.*], nov. 2020. Citado na p. 12.

TECHSPOT. **World's biggest meat processor JBS pays $11 million to ransomware gang**. [*S.l.*], jun. 2021. Disponível em: <https://www.techspot.com/news/90009-world-biggest-meat-processor-jbl-pays-11-million.html>. Acesso em: 28 jun. 2021. Citado na p. 8.

THE ECONOMIC TIMES. **NotPetya: the cyberattack that shook the world**. [*S.l.*], mar. 2022. Disponível em: <https://economictimes.indiatimes.com/tech/newsletters/ettech-unwrapped/notpetya-the-cyberattack-that-shook-the-world/articleshow/89997076.cms?from=mdr>. Acesso em: 12 ago. 2022. Citado na p. 17.

THE HACKER NEWS. **CryptoWall Ransomware raised $325 Million in Revenue for Its Developer**. [*S.l.*], out. 2015. Disponível em: <https://thehackernews.com/2015/10/cryptowall-ransomware.html>. Acesso em: 12 ago. 2022. Citado na p. 16.

UNIT 42. **Highlights from the 2021 Unit 42 Ransomware Threat Report**. [*S.l.*], mar. 2021. Disponível em: <https://unit42.paloaltonetworks.com/ransomware-threat-report-highlights/>. Acesso em: 11 jan. 2022. Citado na p. 15.

ZDNET. **Ransomware: It's a 'golden era' for cyber criminals - and it could get worse before it gets better**. [*S.l.*], out. 2021. Disponível em: <https://www.zdnet.com/article/ransomware-its-a-golden-era-for-cyber-criminals-and-it-could-get-worse-before-it-gets-better/>. Acesso em: 22 jun. 2022. Citado na p. 8.

A ARTIGO

Implementação de um sistema anti-ransomware com abordagem de detecção de cifragem em tempo real

Mateus M. Pereira[1]

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

mateus.p@grad.ufsc.br

Abstract. *Considered the number one virus threat, ransomware attacks have been on the rise over the years. According to a survey carried out by the virtual security company ExtraHop [Canaltech 2022], at least 85% of treaties claimed to have suffered at least one ransomware-type virus attack per year. In its most common form, ransomware encrypts the victim's files, keeping them inaccessible until a ransom is paid, usually in cryptocurrency, affecting everyone from ordinary users to large companies. This article proposes a real-time encryption detection approach for Linux environments capable of detecting a ransomware attempt to encrypt machine data.*

Resumo. *Considerado a ameaça número um de vírus, os ataques de ransomware vem crescendo muito com o passar dos anos. Segundo uma pesquisa desenvolvida pela empresa de segurança virtual ExtraHop [Canaltech 2022], ao menos 85% dos entrevistados afirmaram ter sofrido pelo menos um ataque de vírus do tipo ransomware por ano. Em sua forma mais comum, o ransomware criptografa os arquivos da vítima mantendo-os inacessíveis até o pagamento de um resgate, geralmente em criptomoeda, afetando desde usuários comuns até grandes empresas. Este artigo propõe uma abordagem de detecção de cifragem em tempo real para ambientes Linux capaz de reconhecer uma tentativa do ransomware de criptografar os dados da máquina.*

1. Proposta

Este artigo tem como objetivo propor uma solução em módulo do *kernel Linux* capaz de prevenir cifragem em tempo real. A arquitetura pensada tem como foco expor um ataque do tipo *ransomware* em um ambiente Linux através da verificação de disparidade na modificação de arquivos. O módulo em *kernel* será capaz de determinar quando ele deve agir e impedir que uma modificação a um arquivo seja efetuada. Mais detalhes da implementação estão na seção 2.

É importante ressaltar que a arquitetura proposta tem como objetivo a detecção e prevenção de ataques de *malwares* da classe *Crypto Ransomware*. Essa classe de vírus tem como alvo criptografar os arquivos da vítima e impedir o acesso a eles, entretanto o uso desta solução isolada não contribui para com ataques que visam, por exemplo, sequestrar o MBR da máquina, entre outros ataques que não só bloqueiam ou restringem o acesso ao sistema operacional mas também podem impedir que a máquina inicialize.

2. Desenvolvimento do sistema

Para o desenvolvimento pensou-se em trazer a solução de defesa para dentro do núcleo do próprio sistema operacional, obtendo mais segurança e tornando-se imperceptível ao usuário. Estender o núcleo do *Linux* é possível através do módulo carregável do núcleo (*LKM*), um arquivo objeto que pode ser carregado junto ao núcleo base em execução.

2.1. Desenvolvimento do módulo de *kernel*

O *kernel* é o componente principal do sistema operacional *Linux*. É a principal interface entre o *hardware* de um computador e seus elementos lógicos. Responsável por administrar os recursos, entradas e saídas de dados de cada processo. No *Linux* é possível integrar módulos ao *kernel* em execução incrementando as funcionalidades já existentes e até mesmo criando novas funcionalidades. Neste trabalho três chamadas de sistema foram sobrescritas para incluir os efeitos requeridos pelo fluxo proposto na Figura 1.

2.2. Manipulando eventos

No *kernel* do *Linux* são definidas todas as chamadas essenciais do sistema, como as de leitura e escrita de arquivos. Os endereços de todas essas funções do sistema operacional ficam armazenados em uma tabela estática em uma área de memória protegida dentro do *kernel*. Na versões recentes do *kernel* o endereço dessa tabela não é exposto à outros níveis do sistema, mas é possível encontrá-la com busca por força bruta na memória do *kernel*. Com acesso a tabela é possível modificar os endereços de cada entrada, assim, pode-se modificar as chamadas de sistema apontando-as para endereços distintos do original. Graças a essa liberdade, foi montado uma esquema capaz de controlar os acessos aos arquivos e identificar uma tentativa de criptografar os arquivos do usuário. O esquema pode ser visto na Figura 1.

Quando a máquina, executando o *kernel* modificado, recebe uma chamada de sistema solicitando a abertura de um arquivo, seja de leitura ou escrita, ocorre uma filtragem dos arquivos que devem ser analisados e quais devem ser ignorados. Os parâmetros de filtros podem ser facilmente mudados. Neste projeto o diretório *home* do *Linux* é o escolhido, por ser o diretório que contém a maioria dos arquivos importantes do usuário.

Após a filtragem, um *backup* do arquivo é criado para impedir que um atacante leia os *bytes* de um arquivo em somente leitura e crie uma cópia criptografada dele, excluindo o original. Se a chamada de sistema for de somente leitura o fluxo encerra-se nesta etapa.

Caso a abertura do arquivo implique em uma escrita, o atacante pode sobrescrever os *bytes* originais do arquivo pelos criptografados. Para evitar essa forma de ataque é feito um *hash* baseado na técnica de *hash* sensível à localidade do arquivo aberto e após a chamada de escrita ao sistema operacional os *hashes* do arquivo original são comparados ao modificado para validar se essa é uma escrita incremental ou se há uma tentativa de sobrescrever os *bytes* pelos de um criptografado. Para entender melhor como é feito o *hash* do arquivo veja seção 2.3. Na condição de que seja identificado um ataque a escrita é bloqueada.

Como já citado anteriormente, um *backup* do arquivo é criado sempre que há uma leitura, para garantir que não haja chances de ataque que envolvam deletar o arquivo original e criar uma cópia criptografada. Mas é importante ressaltar que dessa forma os

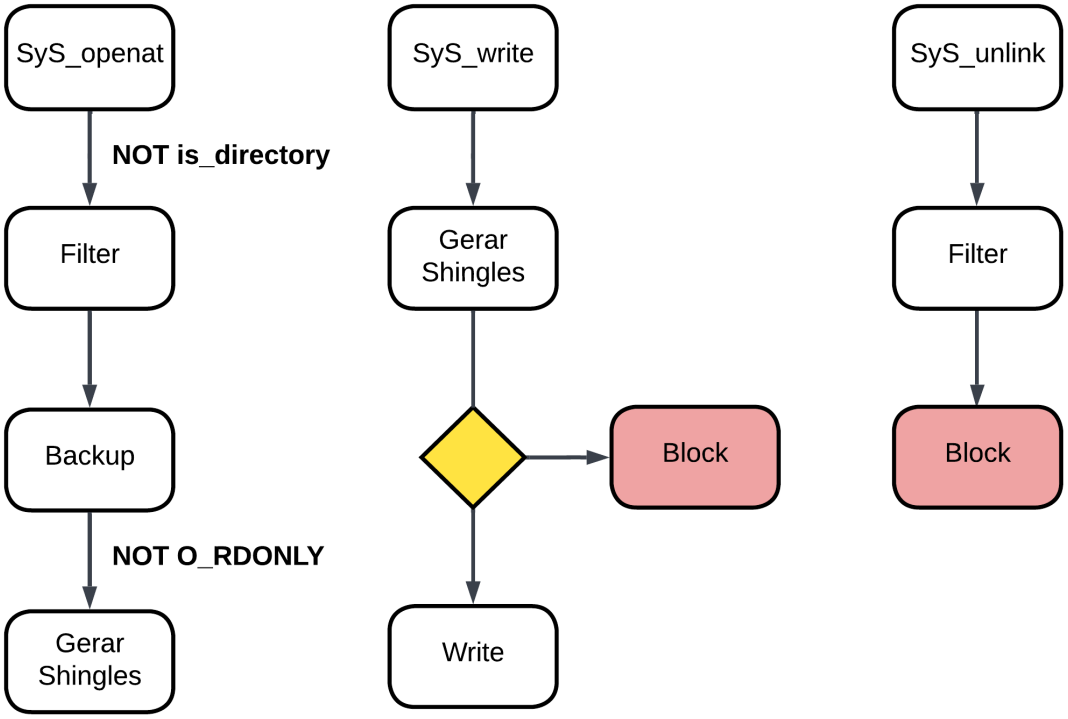

Figure 1. Diagrama da manipulação de chamadas do sistema.

arquivos de *backup* ficariam desprotegidos caso um *ransomware* tentasse apaga-los. Para evitar esse problema houve uma sobrescrita da chamada de sistema *unlink* para bloquear qualquer tentativa de excluir as cópias de segurança.

2.3. Similaridade de *bytes*

Hash sensível à localidade (*LSH*) é uma técnica capaz de produzir *hashes* iguais para entradas de dados semelhantes, a maior aplicabilidade dessa técnica é na procura de vizinhos mais próximos em grandes dimensões [Dasgupta et al. 2011], mas o princípio de seu funcionamento também é aplicável em outras áreas, como na detecção de *spams*. Seguindo essa linha, foi desenvolvido um algoritmo para geração de *hashes* dos arquivos, afim de comparar duas versões do mesmo arquivo e determinar se ele foi criptografado.

Para simplificar a explicação do algoritmo proposto, notação hexadecimal será utilizado no lugar dos bytes do arquivo, mas é importante ressaltar que o algoritmo executado tem como entrada os bytes de um arquivo qualquer. Na figura 2 pode-se ver o funcionamento do algoritmo de geração de *hash*.

Primeiramente os bytes pivôs são selecionados, no caso da Figura 2, **CB** foram escolhidos. Logo então, os bytes são separados em conjuntos delimitados pelo pivô e cada grupo tem seu *hash* gerado, o *hash* de cada conjunto é chamado de *shingle*. Por último, concatena-se todos os *shingles* para construir o *hash* final.

Com um *hash* gerado dessa forma, se o arquivo for alterado, somente parte do *hash* mudará, assim, é possível identificar uma alteração drástica em um arquivo caso os *hashes* sejam completamente diferentes. Por exemplo, mesmo que um arquivo PDF tenha

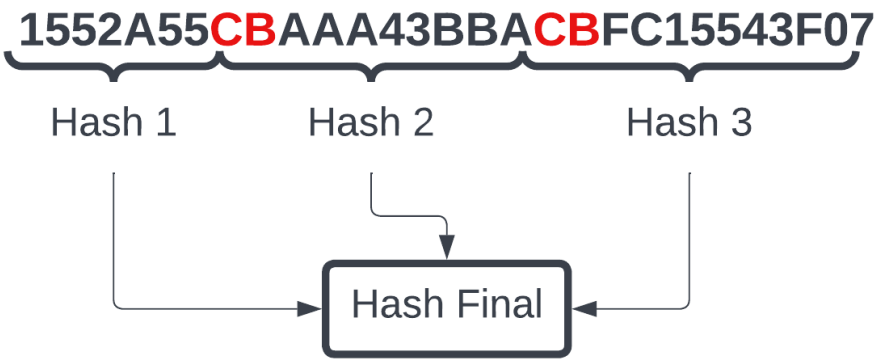

Figure 2. Geração de *Shingles*.

seu texto completamente modificado, essa alteração ainda será válida, já que o arquivo PDF contém toda uma estrutura interna que permanecerá semelhante.

2.4. Testes

Todo desenvolvimento do módulo e os testes foram feitos em réplicas da mesma máquina virtual com sistema operacional *Ubuntu 22.04* com *Kernel* versão *5.15.0*.

Para testar as funcionalidades acrescentadas no *kernel* do *Linux* foram confeccionados dois *scripts* em *Python*. O primeiro atua como um *Crypto Locker* criptografando os arquivos no diretório *home* do usuário. O objetivo desse teste é validar que é possível identificar uma tentativa de criptografar os arquivos do usuário e impedir danos à vítima do ataque *ransomware*. Já o segundo *script* faz leituras e escritas não maliciosas em ativos do usuário, afim de provar que operações legítimas não foram impactadas.

No primeiro teste concluiu-se que os objetivos de proteger os arquivos do usuário contra ataques de *ransomware* que criptografam arquivos da vítima foi alcançado. Já no segundo teste, o objetivo de ser imperceptível ao usuário foi parcialmente alcançado. O módulo do *kernel* trabalha de forma silenciosa não impactando as tarefas do usuário na maior parte do tempo, mas alguns usuários gostam de criptografar seus arquivos e o *kernel* irá bloquear a operação. Outro fator impactante para o usuário, é o fato do arquivo de texto plano possuir uma equivalência direta entre o texto escrito e os bytes do arquivo, fazendo com que a sobrescrita completa de um arquivo de texto simples gere um alarme falso. Para substituir por completo um arquivo de texto é necessário exclui-lo e cria-lo novamente.

Modo	Sem módulo	Com módulo
Leitura	0.41	1.26
Escrita	1.96	4.38

Table 1. Comparação do tempo de execução em milissegundos

Também foram elaborados dois *scripts* para medir o impacto do módulo na performance do sistema. Um deles faz mil leituras consecutivas em um arquivo e calcula o

tempo médio de execução. O outro *script* segue o mesmo padrão, mas desta vez com mil escritas em um arquivo. Pode-se ver o resultado na Tabela 1.

References

Canaltech (2022). *85% das empresas sofrem ao menos um ataque de ransomware por ano, diz estudo*.

Dasgupta, A., Kumar, R., and Sarlos, T. (2011). Fast locality-sensitive hashing. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, page 1073–1081, New York, NY, USA. Association for Computing Machinery.

B REPOSITÓRIO

https://github.com/mateus-mp/TCC

C CÓDIGO

C.1 DECLARAÇÕES

```c
#include <crypto/hash.h>
#include <linux/namei.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/syscalls.h>

MODULE_LICENSE("GPL");

char *filter_list[] = {
    "/home"
};

char *extensions[] = {
    "rar", "zip", "iso", "vcd",
    "csv", "dat", "db", "log",
    "sav", "sql", "tar", "xml",
    "bmp", "gif", "ico", "jpeg",
    "jpg", "png", "svg", "asp",
    "aspx", "css", "htm", "html",
    "js", "php", "py", "c",
    "java", "avi", "mkv", "mp4",
    "mpg", "mpeg", "wmv", "doc",
    "odt", "pdf", "tex", "txt",
    "pem", "crt"
};

typedef asmlinkage int (*syscall_wrapper)(const struct pt_regs *);
syscall_wrapper original_openat;
syscall_wrapper original_write;
syscall_wrapper original_unlinkat;

unsigned long sys_call_table_addr;
```

Código 1 – Declaração de variáveis e tipos

C.2 MD5

```c
static bool md5(const char *data, const size_t len, char **result) {
    struct crypto_shash *shash;
    struct shash_desc *desc;
    size_t size, ret;
    int i;
```

```
    char c[3];
    unsigned char digest[32];

    shash = crypto_alloc_shash("md5", 0, 0);

    if (shash == NULL)
        return false;

    size = sizeof(struct shash_desc) + crypto_shash_descsize(shash);
    desc = kmalloc(size, GFP_KERNEL);
    desc->tfm = shash;

    ret = crypto_shash_digest(desc, data, len, digest);

    for (i = 0; i < 16; i++) {
        sprintf(c, "%02x", digest[i] & 0xFFu);
        memcpy(*result + i * 2, c, 2);
    }

    (*result)[32] = '\0';

    kfree(desc);
    crypto_free_shash(shash);

    return true;
}
```

Código 2 – Geração de *hash* md5

C.3 KADDR_LOOKUP_NAME

```
static unsigned long kaddr_lookup_name(const char *fname_raw)
{
    int i;
    unsigned long kaddr;
    char *fname_lookup, *fname;

    fname_lookup = kvzalloc(NAME_MAX, GFP_KERNEL);
    if (!fname_lookup)
        return 0;

    fname = kvzalloc(strlen(fname_raw) + 4, GFP_KERNEL);
    if (!fname)
        return 0;

    strcpy(fname, fname_raw);
    strcat(fname, "+0x0");
```

```
17
18      kaddr = (unsigned long) &sprint_symbol;
19      kaddr &= 0xfffffffff000000;
20
21      for (i = 0x0 ; i < 0x300000 ; i++) {
22          sprint_symbol(fname_lookup, kaddr);
23          if (strncmp(fname_lookup, fname, strlen(fname)) == 0)
24          {
25              kvfree(fname);
26              kvfree(fname_lookup);
27              return kaddr;
28          }
29
30          kaddr += 0x10;
31      }
32
33      kvfree(fname);
34      kvfree(fname_lookup);
35      return 0;
36  }
```

Código 3 – Procura o endereço de um símbolo

C.4 ENABLE_PAGE_RW

```
1  static void enable_page_rw(void *ptr) {
2      unsigned int level;
3      pte_t *pte = lookup_address((unsigned long) ptr, &level);
4      if(pte->pte & ~_PAGE_RW) {
5          pte->pte |= _PAGE_RW;
6      }
7  }
```

Código 4 – Libera escrita em página

C.5 DISABLE_PAGE_RW

```
1  static void disable_page_rw(void *ptr) {
2      unsigned int level;
3      pte_t *pte = lookup_address((unsigned long) ptr, &level);
4      pte->pte = pte->pte & ~_PAGE_RW;
5  }
```

Código 5 – Bloqueia escrita em página

C.6 GET_ABSOLUTE_PATH_BY_FD

```
static char * get_absolute_path_by_fd(const int fd) {
    char *path = NULL;
    char *page = (char *) __get_free_page(GFP_KERNEL);
    struct path *spath;
    struct file *file = fget(fd);

    if (IS_ERR(file))
        goto exit;

    spath = &file->f_path;
    path_get(spath);
    path = d_path(spath, page, PAGE_SIZE);
    path_put(spath);
    fput(file);

    if (IS_ERR(path))
        path = NULL;

exit:
    free_page((unsigned long) page);
    return path;
}
```

Código 6 – Recupera caminho absoluto pelo *file descriptor*

C.7 GET_ABSOLUTE_PATH_BY_DFD

```
static char * get_absolute_path_by_dfd(const int dfd, const char *filename)
    {
    int flag = 0;
    int ret = -EINVAL;
    unsigned int lookup_flags = 0;
    char *path = NULL;
    char *page = (char *) __get_free_page(GFP_KERNEL);
    struct path spath;

    if ((flag & ~(AT_SYMLINK_NOFOLLOW | AT_NO_AUTOMOUNT)) != 0)
        goto exit;

    if (!(flag & AT_SYMLINK_NOFOLLOW))
        lookup_flags |= LOOKUP_FOLLOW;

    ret = user_path_at(dfd, filename, lookup_flags, &spath);
    if (ret)
```

```
17          goto exit;
18
19      path = d_path(&spath, page, PAGE_SIZE);
20
21  exit:
22      free_page((unsigned long) page);
23      return path;
24  }
```

Código 7 – Recupera caminho absoluto pelo *directory file descriptor*

C.8 STARTSWITH

```
1  static bool startswith(const char *pre, const char *str)
2  {
3      return strncmp(pre, str, strlen(pre)) == 0;
4  }
```

Código 8 – Verifica início da *string*

C.9 FILTER_PATH

```
1  static bool filter_path(const char *path) {
2      int i;
3      int len = sizeof(filter_list)/sizeof(char*);
4
5      for (i = 0; i < len; i++) {
6          if (startswith(filter_list[i], path))
7              return true;
8      }
9
10     return false;
11 }
```

Código 9 – Filtra arquivos para serem analisados

C.10 FILTER_EXTENSION

```
1  static bool filter_extension(char *path) {
2      int i;
3      char *ext = path + strlen(path);
4      size_t len = sizeof(extensions) / sizeof(extensions[0]);
5
6      for (; ext > path; ext--) {
7          if (*ext == '.') {
```

```
8               ext++;
9               break;
10          }
11      }
12
13      for (i = 0; i < len; i++) {
14          if (strcmp(ext, extensions[i]) == 0) {
15              return true;
16          }
17      }
18
19      return false;
20  }
```

Código 10 – Filtra extensões de arquivos a serem analisados

C.11 IS_DELETED

```
1  static bool is_deleted(const char *path) {
2      char d[10];
3      unsigned short index = strlen(path) - 9;
4
5      strncpy(d, &path[index], 9);
6      d[9] = '\0';
7
8      if (strcmp(d, "(deleted)\0") == 0)
9          return true;
10
11     return false;
12 }
```

Código 11 – Verifica se arquivo foi excluído

C.12 IS_DIRECTORY

```
1  static bool is_directory(const char *path) {
2      int error;
3      struct path spath;
4      struct inode *inode;
5
6      error = kern_path(path, LOOKUP_FOLLOW, &spath);
7      if (error)
8          goto error;
9
10     inode = spath.dentry->d_inode;
11
```

```
12      return S_ISDIR(inode->i_mode);
13
14  error:
15      return true;
16  }
```

Código 12 – Verifica se caminho é para um diretório

C.13 GENERATESHINGLES

```
1   static unsigned short generateShingles(char *buffer, const size_t
        bufferSize, const char *pivot, char ***shingles, unsigned short **
        shinglesSizes) {
2       size_t i;
3       unsigned short shingleIndex, counter;
4
5       shingleIndex = 1;
6       counter = 0;
7       *shingles = kmalloc(sizeof(char *), GFP_KERNEL);
8       *shinglesSizes = kmalloc(sizeof(unsigned short), GFP_KERNEL);
9       (*shingles)[0] = buffer;
10
11      for (i = 2; i < bufferSize - 1; i++) {
12          if (shingleIndex > 49)
13              break;
14
15          if (buffer[i] == pivot[0] && buffer[i + 1] == pivot[1]) {
16              *shinglesSizes = krealloc(*shinglesSizes, shingleIndex * sizeof
        (unsigned short), GFP_KERNEL);
17              (*shinglesSizes)[shingleIndex - 1] = i - counter;
18              *shingles = krealloc(*shingles, (shingleIndex + 1) * sizeof(
        char *), GFP_KERNEL);
19              (*shingles)[shingleIndex++] = &buffer[i];
20
21              counter = i++;
22          }
23      }
24
25      *shinglesSizes = krealloc(*shinglesSizes, shingleIndex * sizeof(
        unsigned short), GFP_KERNEL);
26      (*shinglesSizes)[shingleIndex - 1] = bufferSize - counter - 1;
27
28      return shingleIndex;
29  }
```

Código 13 – Geração de *Shingles*

C.14 CREATEHASH

```
static char *createHash(char *path) {
    char *result;

    result = kmalloc(33, GFP_KERNEL);
    result[32] = '\0';

    if (!md5(path, strlen(path), &result)) {
        kfree(result);
        return NULL;
    }

    return result;
}
```

Código 14 – Cria *hash* do caminho do arquivo

C.15 SAVEFILE

```
static void saveFile(const char *hash, const char * write, const size_t
    writeSize) {
    char hashPath[38];
    struct file *writeFile;
    size_t size;

    hashPath[37] = '\0';

    sprintf(hashPath, "/tmp/%s", hash);

    writeFile = filp_open(hashPath, O_CREAT | O_WRONLY, 0777);
    writeFile->f_pos = 0;
    size = kernel_write(writeFile, write, 32 * writeSize + 4, &writeFile->
    f_pos);

    filp_close(writeFile, NULL);
}
```

Código 15 – Salva arquivos temporários

C.16 CREATEHASHES

```
static bool createHashes(const char *pivot, char ***shingles, unsigned
    short **shinglesSizes, const unsigned short totalShingles, char **write,
     size_t *writeSize) {
    size_t i, size;
```

```
3     char *input;
4     char *result;
5
6     result = kmalloc(33, GFP_KERNEL);
7
8     size = (*shinglesSizes)[0];
9     input = kmalloc(size, GFP_KERNEL);
10    input[size - 1] = '\0';
11    strncpy(input, (*shingles)[0], size);
12
13    if (!md5(input, size, &result))
14        goto free;
15
16    *write = kmalloc(36, GFP_KERNEL);
17    (*write)[0] = pivot[0];
18    (*write)[1] = pivot[1];
19    (*write)[2] = ' ';
20    (*write)[35] = '\0';
21    sprintf(&(*write)[3], "%s", result);
22
23    for (i = 1; i < totalShingles; i++) {
24        size = (*shinglesSizes)[i];
25        input = krealloc(input, size, GFP_KERNEL);
26        input[size - 1] = '\0';
27        strncpy(input, (*shingles)[i], size);
28
29        if (!md5(input, size, &result))
30            goto free;
31
32        *write = krealloc(*write, 32 * (i + 1) + 4, GFP_KERNEL);
33        (*write)[32 * (i + 1)] = '\0';
34        sprintf(*write, "%s %s", *write, result);
35    }
36
37    *writeSize = i + 1;
38
39    kfree(input);
40    kfree(result);
41
42    return true;
43
44 free:
45    kfree(input);
46    kfree(result);
47
```

```
48     return false;
49 }
```

Código 16 – Gera *hashes* a partir dos *Shingles*

C.17 STRDUP

```
1  static char * strdup(const char *src) {
2      size_t len = strlen(src);
3      char *s = kmalloc(len, GFP_KERNEL);
4
5      if (s == NULL)
6          return NULL;
7
8      return (char *)memcpy(s, src, len);
9  }
```

Código 17 – Cria cópia de *string* na memória

C.18 GET_FILE_SIZE

```
1  static long get_file_size(const char *path) {
2      struct file *file;
3      long size;
4
5      file = filp_open(path, O_RDONLY | O_LARGEFILE, 0777);
6
7      if (IS_ERR(file))
8          return -1;
9
10     vfs_llseek(file, 0, SEEK_SET);
11     size = vfs_llseek(file, 0, SEEK_END);
12     vfs_llseek(file, 0, SEEK_CUR);
13
14     filp_close(file, NULL);
15
16     return size;
17 }
```

Código 18 – Retorna tamanho do arquivo

C.19 READ_FILE

```
1  static size_t read_file(const char *path, const size_t size, char **buffer)
       {
```

```
    struct file *file;
    size_t ret;

    file = filp_open(path, O_RDONLY | O_LARGEFILE, 0777);

    if (IS_ERR(file)) {
        *buffer = NULL;
        return 0;
    }

    file->f_pos = 0;

    *buffer = kmalloc(size + 1, GFP_KERNEL);
    ret = kernel_read(file, *buffer, size, &file->f_pos);
    (*buffer)[ret] = '\0';

    filp_close(file, NULL);

    return ret;
}
```

Código 19 – Leitura de arquivo

C.20 WRITE_FILE

```
static void write_file(const char *path, const char *buffer, const size_t
    size, const int flags) {
    struct file *file;

    file = filp_open(path, flags, 0777);

    file->f_pos = 0;
    kernel_write(file, buffer, size, &file->f_pos);

    filp_close(file, NULL);
}
```

Código 20 – Escrita de arquivo

C.21 HOOKED_OPENAT

```
static asmlinkage int hooked_openat(const struct pt_regs *regs) {
    struct file *file;
    size_t ret, pivotIndex, writeSize, len;
    long size;
    char hashPath[41];
```

```
6     char pivot[3];
7     char *buffer;
8     char *write;
9     char *tpath;
10    char **shingles;
11    unsigned short *shinglesSizes;
12    unsigned short totalShingles;
13    char *hash;
14    struct inode *parent_inode;
15    char *path = (char *) regs->si;
16
17    if (path[0] != '/') {
18        int dfd = regs->di;
19
20        path = get_absolute_path_by_dfd(dfd, path);
21        if (!path)
22            goto exit;
23    }
24
25    if (is_directory(path))
26        goto exit;
27
28    if (!filter_path(path))
29        goto exit;
30
31    if (!filter_extension(path))
32        goto exit;
33
34    len = strlen(path);
35    tpath = kmalloc(len + 5, GFP_KERNEL);
36    tpath[len+4] = '\0';
37    sprintf(tpath, "%s.tcc", path);
38
39    size = get_file_size(tpath);
40
41    if (size < 0)
42        goto notcc;
43
44    ret = read_file(tpath, size, &buffer);
45
46    if (!buffer)
47        goto notcc;
48
49    write_file(path, buffer, ret, O_RDWR | O_CREAT | O_TRUNC | O_LARGEFILE)
      ;
50
51    file = filp_open(tpath, O_RDONLY | O_LARGEFILE, 0777);
52
```

```
53      parent_inode = file->f_path.dentry->d_parent->d_inode;
54      inode_lock(parent_inode);
55      vfs_unlink(&init_user_ns, parent_inode, file->f_path.dentry, NULL);
56      inode_unlock(parent_inode);
57
58      filp_close(file, NULL);
59
60      kfree(buffer);
61
62  notcc:
63      kfree(tpath);
64
65      hash = createHash(path);
66
67      if (hash == NULL)
68          goto exit;
69
70      hashPath[40] = '\0';
71      sprintf(hashPath, "/backup/%s", hash);
72
73      size = get_file_size(path);
74
75      if (size < 3 || size > 524288000)
76          goto free2;
77
78      ret = read_file(path, size, &buffer);
79
80      if (!buffer)
81          goto free2;
82
83      write_file(hashPath, buffer, ret, O_RDWR | O_CREAT | O_TRUNC |
        O_LARGEFILE);
84
85      if ((regs->dx&O_ACCMODE) == O_RDONLY)
86          goto free3;
87
88      pivotIndex = ret / 2;
89
90      do {
91          if (pivotIndex >= ret - 2) {
92              pivot[0] = 'a';
93              pivot[1] = 'a';
94              break;
95          }
96          pivot[0] = buffer[pivotIndex];
97          pivot[1] = buffer[pivotIndex + 1];
98          pivotIndex += 2;
99      } while (pivot[0] == ' ' || pivot[1] == ' ');
```

```
100     pivot[2] = '\0';
101
102     totalShingles = generateShingles(buffer, ret, pivot, &shingles, &
        shinglesSizes);
103
104     if (!createHashes(pivot, &shingles, &shinglesSizes, totalShingles, &
        write, &writeSize))
105         goto free;
106
107     saveFile(hash, write, writeSize);
108
109 free:
110     kfree(shingles);
111     kfree(write);
112     kfree(shinglesSizes);
113
114 free3:
115     kfree(buffer);
116
117 free2:
118     kfree(hash);
119
120 exit:
121     return (*original_openat)(regs);
122 }
```

Código 21 – Novo *openat*

C.22 COMPAREHASHES

```
1  static bool compareHashes(const char *oldHashTxt, const char *newHashTxt) {
2      char *token1;
3      char *token2;
4      char *copy1, *r1;
5      char *copy2, *r2;
6
7      copy1 = r1 = strdup(oldHashTxt);
8
9      token1 = strsep(&copy1, " ");
10
11     while(token1 != NULL) {
12         token1 = strsep(&copy1, " ");
13
14         if (token1 == NULL)
15             break;
16
17         copy2 = r2 = strdup(newHashTxt);
```

```
        token2 = strsep(&copy2, " ");

        while(token2 != NULL) {
            token2 = strsep(&copy2, " ");

            if (token2 == NULL)
                break;

            if (strcmp(token1, token2) == 0) {
                kfree(r1);
                kfree(r2);
                return true;
            }
        }
        kfree(r2);
    }

    kfree(r1);
    return false;
}
```

Código 22 – Compara *hashes* de dois arquivos

C.23 HOOKED_UNLINKAT

```
static asmlinkage int hooked_unlinkat(const struct pt_regs *regs) {
    char *path = (char *) regs->si;

    if (path[0] != '/') {
        int dfd = regs->di;

        path = get_absolute_path_by_dfd(dfd, path);
        if (!path)
            goto exit;
    }

    if (startswith("/backup", path))
        goto block;

exit:
    return (*original_unlinkat)(regs);

block:
    return (-EACCES);
}
```

Código 23 – Novo *unlinkat*

C.24 HOOKED_WRITE

```
static asmlinkage int hooked_write(const struct pt_regs *regs) {
    struct file *file;
    int fd = regs->di;
    size_t ret, writeSize, len;
    long size, bsize;
    char pivot[3];
    char hashPath[38];
    char backPath[41];
    char *result;
    char *hashTxt;
    char *write;
    char *tpath;
    char *buffer;
    char **shingles;
    unsigned short *shinglesSizes;
    unsigned short totalShingles;
    char *path = get_absolute_path_by_fd(fd);
    char *bpath = strdup(path);

    if (!path)
        goto exit;

    if (is_deleted(path))
        goto exit;

    if (is_directory(path))
        goto exit;

    if (!filter_path(path))
        goto exit;

    if (!filter_extension(path))
        goto exit;

    file = fget(fd);

    if (IS_ERR(file))
        goto exit;

    vfs_llseek(file, 0, SEEK_SET);
    size = vfs_llseek(file, 0, SEEK_END);
    vfs_llseek(file, 0, SEEK_CUR);

    result = kmalloc(33, GFP_KERNEL);

```

```
46      if (!md5(path, strlen(path), &result)) {
47          fput(file);
48          goto free2;
49      }
50
51      fput(file);
52
53      backPath[40] = '\0';
54      sprintf(backPath, "/backup/%s", result);
55
56      bsize = get_file_size(backPath);
57
58      if (bsize < 0)
59          goto free2;
60
61      if (size > 0 && regs->dx + size <= bsize)
62          goto free2;
63
64      if (regs->dx <= bsize)
65          goto free2;
66
67      hashPath[37] = '\0';
68      sprintf(hashPath, "/tmp/%s", result);
69
70      size = get_file_size(hashPath);
71
72      if (size < 0)
73          goto free2;
74
75      ret = read_file(hashPath, size, &hashTxt);
76
77      if (!hashTxt)
78          goto free2;
79
80      pivot[0] = hashTxt[0];
81      pivot[1] = hashTxt[1];
82      pivot[2] = '\0';
83
84      totalShingles = generateShingles((char *)regs->si, regs->dx + 1, pivot,
        &shingles, &shinglesSizes);
85
86      if (!createHashes(pivot, &shingles, &shinglesSizes, totalShingles, &
       write, &writeSize))
87          goto free1;
88
89      if (!compareHashes(hashTxt, write)) {
90          kfree(shingles);
91          kfree(hashTxt);
```

```
92          kfree(write);
93          kfree(shinglesSizes);
94          goto block;
95      }
96
97  free1:
98      kfree(shingles);
99      kfree(hashTxt);
100     kfree(write);
101     kfree(shinglesSizes);
102
103 free2:
104     kfree(result);
105
106 exit:
107     kfree(bpath);
108     return (*original_write)(regs);
109
110 block:
111     ret = read_file(backPath, bsize, &buffer);
112
113     if (!buffer)
114         goto exit2;
115
116     len = strlen(bpath);
117     tpath = kmalloc(len + 5, GFP_KERNEL);
118     tpath[len+4] = '\0';
119     sprintf(tpath, "%s.tcc", bpath);
120
121     write_file(tpath, buffer, ret, O_RDWR | O_CREAT | O_TRUNC | O_LARGEFILE
        );
122
123     kfree(buffer);
124     kfree(tpath);
125
126 exit2:
127     kfree(bpath);
128     kfree(result);
129
130     return (-EACCES);
131 }
```

Código 24 – Novo *write*

C.25 SWAP_SYSCALL

```
1 static int swap_syscall(unsigned short nr, syscall_wrapper *old,
```

```
    syscall_wrapper new) {
    *old = ((syscall_wrapper *)sys_call_table_addr)[nr];
    if (!*old)
        return 1;

    ((syscall_wrapper *)sys_call_table_addr)[nr] = new;

    return 0;
}
```

Código 25 – Troca endereços na tabela de chamadas de sistema

C.26 START

```
static int __init start(void) {
    printk(KERN_INFO "Anti-Ransomware Module has been started.\n");

    sys_call_table_addr = kaddr_lookup_name("sys_call_table");
    printk("SYS_CALL_TABLE_ADDR: %lu.\n", sys_call_table_addr);
    if (!sys_call_table_addr)
        goto error;

    enable_page_rw((void *)sys_call_table_addr);

    if (swap_syscall(__NR_openat, &original_openat, hooked_openat))
        goto error;
    printk(KERN_INFO "Openat got hooked.\n");

    if (swap_syscall(__NR_write, &original_write, hooked_write))
        goto error;
    printk(KERN_INFO "Write got hooked.\n");

    if (swap_syscall(__NR_unlinkat, &original_unlinkat, hooked_unlinkat))
        goto error;
    printk(KERN_INFO "Unlinkat got hooked.\n");

    disable_page_rw((void *)sys_call_table_addr);

    return 0;

error:
    disable_page_rw((void *)sys_call_table_addr);
    printk(KERN_ERR "An error occurred during module initialization.\n");
    return 1;
}
```

Código 26 – Inicializa módulo

C.27 STOP

```
1 static void __exit stop(void) {
2     enable_page_rw((void *)sys_call_table_addr);
3     ((syscall_wrapper *)sys_call_table_addr)[__NR_openat] = original_openat
      ;
4     ((syscall_wrapper *)sys_call_table_addr)[__NR_write] = original_write;
5     ((syscall_wrapper *)sys_call_table_addr)[__NR_unlinkat] =
      original_unlinkat;
6     disable_page_rw((void *)sys_call_table_addr);
7
8     printk(KERN_INFO "Anti-Ransomware Module has been deactivated.\n");
9 }
```

Código 27 – Finaliza execução do módulo

C.28 DEFINIÇÃO DE INICIALIZAÇÃO E FINALIZAÇÃO

```
1 module_init(start);
2 module_exit(stop);
```

Código 28 – Define função de inicialização e finalização do módulo de *kernel*