Melhorias no Processo de Armazenamento de Dados em TPM para Gerenciamento de Integridade

Mariane M. S. Zeitouni[1], **Marcela Tassyany Galdino Santos**[1], **Reinaldo Cezar De Morais Gomes**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Campina Grande (UFCG)

mariane.zeitouni@ccc.ufcg.edu.br, {marcela,reinaldo}@computacao.ufcg.edu.br

Abstract. *Some electronic devices have native solutions to guarantee their integrity, one example is the TPM (Trusted Platform Module), a chip dedicated to security. In virtual machines, a vTPM (Virtual Trusted Platform Module) can be found, which, when anchored with the TPM, can take advantage of the security robustness that the TPM has. However, there is an obstacle to this strategy, and that is where the objective of this work arises. The vTPM, when generating multiple requests to the TPM, can generate an overload on the chip and, to solve this as well as multiple requests not necessarily coming from the vTPM, the work proposes the implementation of a request scheduler.*

Resumo. *Alguns dispositivos eletrônicos possuem soluções nativas para garantir sua integridade, um exemplo é o TPM (Trusted Platform Module), um chip dedicado à segurança. Nas máquinas virtuais, um vTPM(Virtual Trusted Platform Module) pode ser encontrado, este, quando ancorado com o TPM pode usufruir da robustez de segurança que o TPM possui. Entretanto existe um obstáculo nessa estratégia e é onde surge o objetivo deste trabalho. O vTPM, ao gerar múltiplas requisições ao TPM, pode gerar uma sobrecarga no chip e para solucionar isso bem como múltiplas requisições não necessariamente vindas do vTPM, o trabalho propõe a implementação de um gerenciador de requisições.*

1. Introdução

De acordo com [Arthur et al. 2015], o chip TPM (*Trusted Platform Module*) é um co-processador criptográfico presente na maioria dos PCs e servidores. Este chip foi desenvolvido em resposta ao avanço significativo da tecnologia e ao impacto transformador da internet no mercado de computadores pessoais. Segundo os autores, a segurança dos hardwares e softwares não recebeu a devida atenção, pois a prioridade era a facilidade de uso dos dispositivos e suas soluções.

O TPM é capaz de realizar operações criptográficas, como assinatura digital, e gerar chaves simétricas e assimétricas. Além disso, o dispositivo pode certificar essas chaves com base em sua chave de endosso. A chave de endosso do TPM e as chaves por ele geradas não podem ser extraídas do dispositivo, garantindo proteção contra sistemas operacionais comprometidos e administradores de sistemas com privilégios elevados. Porém, os TPMs de hardware são reconhecidos por possuírem recursos limitados, como

uma quantidade reduzida de NVRAM (*Non-Volatile Random Access Memory*), baixo desempenho de entrada e saída, e um número restrito de PCRs (*Platform Configuration Registers*) [Arthur and Challener 2015].

A virtualização das máquinas está se tornando mais comum devido a fatores como a flexibilidade, a escalabilidade e a eficiência no uso de recursos que ela proporciona. No entanto, as máquinas virtuais não possuem o chip TPM, e para resolver essa limitação, surgiu o vTPM (*Virtual Trusted Platform Module*), que é a virtualização do chip TPM. O vTPM emula as funcionalidades de um TPM físico, permitindo que máquinas virtuais se beneficiem das mesmas capacidades de segurança que as máquinas físicas, garantindo assim um ambiente mais seguro. Contudo, o vTPM é mais suscetível a ataques de *software*, então uma solução para que uma máquina virtual desfrute de toda a segurança do TPM é a ancoragem entre os dois - vTPM e TPM. Enquanto o vTPM realiza requisições de forma serial ao chip, este atenderá essas chamadas. Com esse exemplo de múltiplas requisições chegando no TPM, pode-se evidenciar as limitações em termos de capacidade de processamento e suporte do chip.

Uma possível solução, conforme apresentado no artigo dos autores [Eckel et al.], é a implementação de um gerenciador de requisições. Esse gerenciador visa solucionar o gargalo causado por múltiplas chamadas e evitar que o chip entre em um estado bloqueado. Com essa solução implementada, espera-se que o TPM registre as devidas informações das requisições feita pelo gerenciador, evitando assim que o chip entre em um estado incapaz de realizar as devidas validações que garantem a segurança das máquinas.

Para solucionar esse problema, foi implementado um algoritmo que tem como objetivo gerenciar as chamadas para o chip, fazendo com que este não fique sobrecarregado com múltiplas requisições recebidas, sejam elas de VMs ou não. A implementação faz uso da biblioteca Go-TPM, a qual se comunica com o TPM da máquina e o algoritmo implementado em Golang faz uso de canais para que as requisições sejam enfileiradas em ordem de chegada. Os resultados dos testes realizados foram satisfatórios comprovando que a implementação cumpriu com o seu objetivo.

A continuação deste trabalho está organizada da seguinte forma:

- Seção 2: Fundamentação Teórica, onde serão descritos alguns conceitos importantes para o entendimento deste trabalho.
- Seção 3: Trabalhos Relacionados, apresentando pesquisas e implementações anteriores relevantes para o contexto deste estudo.
- Seção 4: Metodologia, detalhando como a implementação foi realizada, incluindo as técnicas e ferramentas utilizadas.
- Seção 5: Resultados e Discussões, onde serão apresentados e analisados os resultados obtidos com a implementação.
- Seção 6: Conclusão e Sugestão de Trabalhos Futuros, finalizando com as considerações finais e possíveis direções para pesquisas futuras.

2. Fundamentação Teórica

Nesta seção, serão abordados os conceitos fundamentais relacionados ao *Trusted Platform Module* (TPM). A compreensão desses conceitos é crucial para a análise das técnicas de segurança utilizadas em ambientes de computação moderna.

2.1. O *Trusted Platform Module*

O TPM é um padrão de segurança desenvolvido pelo *Trusted Computing Group*. Ele é integrado fisicamente como um chip na placa-mãe de um sistema e é gerenciado por software através de comandos específicos. Suas funcionalidades incluem operações criptográficas como criação de chaves assimétricas, encriptação, decriptação, assinatura digital e transferência de chaves entre TPMs, além de geração de números aleatórios e hashing. O TPM também oferece um meio seguro para armazenar informações sensíveis, como chaves criptográficas. Devido à sua implementação em hardware, o TPM é projetado para ser resistente a ataques de software [Trusted Computing Group].

Um aspecto que é de interesse para o atual trabalho é o Registrador de Configuração da Plataforma (PCR, do inglês *Platform Configuration Register*) que faz parte da verificação de integridade do ambiente de execução com o chip. Estes PCRs são registros especiais usados pelo TPM para armazenar valores de hash criptograficamente seguros que representam o estado de diferentes componentes do sistema em um determinado momento.

O principal objetivo dos PCRs é registrar e monitorar mudanças na configuração do sistema. Os PCRs são inicializados quando o TPM é ativado e só podem ser modificados por meio de reset ou extensão. O TPM possui 24 registradores (PCR) onde são guardadas as medições, no entanto, cada medição não é atribuída a um PCR diretamente. Para incluir uma medição, o TPM realiza uma operação de hash estendido sobre o último valor do PCR. Por exemplo, se o PCR 10 for estendido com alguma medição M, então o novo valor do PCR 10 é computado da seguinte maneira: $PCR10 = hash(PCR10 + M)$ [Tassyany et al. 2021]. Isso garante que cada operação de extend produza um valor de PCR que depende de todos os valores anteriores que foram estendidos, fazendo assim uma cadeia de integridade, a qual cria uma dificuldade em falsificar ou manipular sem que seja detectada.

A verificação de autenticidade de um TPM envolve o uso de certificados digitais emitidos por uma autoridade certificadora (AC) e EK (*Endorsement Key*). A EK é uma chave assimétrica, e não é possível extrair sua parte privada do TPM, enquanto a parte pública pode ser obtida a partir do certificado X509 injetado na RAM não-volátil. Então, para solicitar um desafio, basta que um cliente envie o certificado X509 referente à EK. Após extrair a EK pública do certificado x509, o verificador analisa sua autenticidade checando se a mesma foi assinada por alguma AC confiável. A assinatura é verificada usando certificados das ACs, se essa verificação for bem sucedida, então o verificador sabe que esta é a parte pública de uma EK cuja parte privada nunca pode ser extraída de um TPM. Portanto, o verificador criptografa algum segredo aleatório, e apenas o cliente com o TPM que possui a parte privada daquela EK será capaz de descriptografar o segredo. Finalmente, o desafio é concluído quando o cliente apresenta o segredo em texto plano ao verificador. [Tassyany et al. 2021]

3. Trabalhos Relacionados

Algumas pesquisas encontradas tratam de tópicos relevantes para este trabalho. No artigo dos autores [Perez et al.2006] destaca-se a importância de um sistema de segurança em um ambiente virtual. É discutida a criação e implementação do vTPM em uma máquina virtual, permitindo que ela se beneficie de um nível de segurança mais robusto, similar

ao proporcionado pelo TPM, além de que, o TPM não possui recursos suficientes para lidar com várias VMs. Para que esse nível de segurança seja completo, é necessário que o vTPM seja ancorado ao TPM físico.

Uma lacuna da pesquisa dos autores citados anteriormente é o fato de que se várias máquinas virtuais estão usando o mesmo chip TPM, então este chip ficará sobrecarregado em algum momento quando receber múltiplas requisições de diversas máquinas virtuais. Como foi explicado anteriormente, essa solução de ancoragem evidencia a limitação de recursos do TPM, e neste cenário que o gerenciador de requisições se encaixa.

No artigo de [Eckel et al. 2021] é proposto melhorias no registro de eventos no TPM para aplicações de sistemas operacionais. Introduz uma biblioteca de software e um serviço de sistema para gerenciar o acesso concorrente ao TPM, além de consolidar múltiplos eventos em um único evento TPM para permitir um registro contínuo e ordenado. Apresenta também uma prova de conceito (PoC) que integra a biblioteca no ambiente JVM para medir classes Java e se comunica com o serviço do sistema, com o código fonte disponível publicamente. O estudo inclui medições de desempenho para validar a implementação e analisa as implicações para processos de atestação remota.

No entanto, ainda existe uma lacuna significativa: a ausência de um mecanismo eficaz para evitar que o TPM trave sob alta demanda de requisições. A solução apresentada neste trabalho preenche essa lacuna ao introduzir um gerenciador de requisições, que mantém o TPM funcionando de maneira eficiente e ininterrupta, independentemente do volume de requisições. Isso garante um desempenho estável e contínuo do TPM, melhorando a confiabilidade e a robustez do sistema proposto por [Eckel et al].

A pesquisa dos autores [Schmitz et al. 2011] tem como um dos objetivos principais desenvolver um conjunto de ferramentas e benchmarks para avaliar o desempenho do TPM em uma variedade de cenários realistas e opções de design. Um dos focos é criar mini-benchmarks simples que exercitem os principais serviços do TPM, permitindo avaliações isoladas e combinadas desses serviços, especialmente em ambientes multicore. Além disso, os autores investigaram otimizações de desempenho, como o reordenamento de requisições do TPM e o uso de múltiplos TPMs, demonstrando melhorias significativas no desempenho conforme aumenta o número de aplicações concorrentes que utilizam o TPM.

O estudo de [Schmitz et al. 2011] possui muitos pontos que agregam este trabalho. A pesquisa mostra que foi implementado um algoritmo que leva em consideração os ciclos que determinada requisição irá precisar fazer para ficar pronta, ou seja, a ”menor”requisição ganha a vez na fila. Esse algoritmo tem o propósito de não deixar que requisições mais rápidas de serem respondidas fiquem atrás de requisições mais caras de serem atendidas. Apesar da solução obter resultados positivos, os autores não comentam no volume de requisições que a solução pode lidar sem que o TPM trave e os resultados obtidos mostram volumes de requisições consideravelmente menores do que irão ser mostrados ao final deste trabalho.

4. Metodologia

Nesta seção, será abordada a metodologia utilizada para solucionar a problemática proposta. Primeiramente, discutiremos uma visão macro do gerenciador. Em seguida, detalharemos as funções específicas implementadas em Golang, que foram desenvolvidas

para executar operações, como a escalonagem das requisições, garantindo a integridade e segurança dos dados. Exploraremos as características dessas funções, sua interação com o TPM, e como foram empregadas técnicas de sincronização para prevenir condições de corrida e assegurar a execução correta das operações em um ambiente concorrente.

4.1. Gerenciador de Requisições

O gerenciador de requisições proposto neste trabalho lida com múltiplas requisições. As operações implementadas fazem leituras das entradas, chamadas da biblioteca Go-TPM e escrita dos resultados em arquivos para depuração. A implementação tem o design projetado para suportar qualquer número de requisições, o que auxilia o TPM a não entrar em um estado bloqueado. Este design robusto e escalável garante que o sistema possa gerenciar de forma eficiente um alto volume de operações simultâneas, mantendo a integridade e a segurança das transações. Além disso, ao utilizar *goroutines* para paralelizar a leitura e o processamento das entradas, o gerenciador maximiza a utilização dos recursos do sistema, reduzindo o tempo de resposta e aumentando a confiabilidade geral do serviço.

4.2. Avaliação Experimental

A biblioteca go-tpm, utilizada na solução proposta, foi desenvolvida em Golang e tem como objetivo se comunicar diretamente com o TPM em máquinas Linux ou Windows. Embora a biblioteca não possua todas as funcionalidades do TPM 1.2 ou 2.0, ela oferece inúmeras funções que interagem com o chip nessas versões. No experimento realizado, a versão do chip 2.0 foi utilizada com o objetivo de realizar leitura e *extend* do PCR. As funções correspondentes a essas operações são *ReadPCR()* e *PCRExtend()*.

A função *PCRExtend*[1] é utilizada para estender um valor PCR em um TPM, ela recebe como entradas um objeto que implementa a interface *io.ReadWriter* para comunicação com o chip, um identificador de PCR, um algoritmo de *hash*, o valor do *hash* a ser estendido e uma senha opcional. A função constrói o comando de extensão do PCR e o envia ao TPM, retornando qualquer erro encontrado durante o processo.

A função *ReadPCR*[1] é usada para ler o valor de um PCR específico em um TPM. Ela recebe como entradas um objeto que implementa a interface *io.ReadWriter* para comunicação com o TPM, um índice de PCR e um algoritmo de *hash*. A função cria uma seleção de PCR, lê os valores dos PCRs do TPM e retorna o valor do PCR especificado. Se ocorrer algum erro ou se o valor do PCR não estiver presente na resposta, a função retorna um erro apropriado.

O algoritmo implementado em Go tem como objetivo gerenciar chamadas ao TPM para gerenciar eficientemente múltiplas requisições simultâneas, garantindo a segurança e a integridade das operações criptográficas. Utilizando a linguagem Go, conhecida por sua simplicidade e excelente suporte para concorrência, o algoritmo aproveita a biblioteca citada anteriormente para facilitar a comunicação com o TPM.

O algoritmo possui quatro funções: *main()*, *read()*, *processInput()* e *compareOutput()*. A função principal, *main()*, do programa executa um processo em que se estabelece uma conexão com o TPM, faz a criação de dois canais de comunicação para gerenciar requisições de entrada e sinalização de conclusão. Utilizando *goroutines*, a função simultaneamente lê as entradas e as processa. Após a conclusão das operações de leitura

[1]https://github.com/google/go-tpm/blob/main/legacy/tpm2/tpm2.go

e processamento, os buffers de escrita são esvaziados para garantir que todos os dados foram registrados corretamente. Finalmente, o tempo total de execução do programa é calculado e exibido, permitindo uma análise de desempenho.

A função *read()* recebe dois parâmetros de entrada, o canal que receberá as requisições e outro canal de tipo booleano para indicar que não receberá mais entradas. A funcionalidade principal dessa função é ler o arquivo de entrada que vai simular a chegada das requisições e enviá-las para o canal. Este arquivo de entrada contém os valores dos PCRs a serem estendidos. Quando todas as entradas forem enviadas para o canal, uma *flag* com o valor *true* é inserida para sinalizar que não haverão mais entradas a serem lidas.

A função *processInput()* vai operar sob os valores recebidos pelo canal. Ao receber as entradas, as funções de *read* e *extend* da biblioteca Go-TPM irão ser chamadas para realizar as suas devidas funções e em seguida, as saídas serão escritas em dois arquivos - output.txt e log.txt. Esses arquivos irão ser utilizados para fins comparativos e de depuração de erros.

A função compareOutput() compara dois arquivos, linha a linha, para a validação da operação. Os arquivos possuem a saída gerada pela operação realizada na função anterior(output.txt) e no expected.txt tem-se os resultados gerados anteriormente sem o uso do algoritmo implementado. Dessa forma, consegue-se validar que o uso do gerenciador não interfere no resultado esperado e cumpre com o objetivo de escalonar as chamadas sem interferir nos resultados das operações.

O ambiente de testes foi estabelecido em um computador equipado com um processador Intel i9-12900 e 32 GB de RAM. Foi utilizado o NPCT75x TPM 2.0 como modelo de TPM de hardware. O sistema operacional foi o Ubuntu 20.04, operando com a versão 4.2.1 do QEMU. Para testes, configurou-se uma máquina virtual operando com OpenSUSE 5.14, 1 vCPU e 1,5 GB de memória principal. Esta VM dispõe de um vTPM emulado pelo SWTPM 0.8.1, usando a biblioteca libtpms versão 0.9.6.

5. Resultados e Discussões

A função *compareOutput()* teve como objetivo comparar os dados resultantes da função *processInput()*, onde ocorre o gerenciamento das requisições ao TPM, com os dados esperados. Esses dados esperados foram coletados previamente a partir de requisições reais ao TPM e sem o uso de nenhuma biblioteca específica. A função *compareOutput()* realizou uma comparação minuciosa entre os dados reais e os gerados pela implementação proposta, verificando a precisão e a consistência dos resultados. Essa comparação confirmou que a implementação produz resultados iguais aos obtidos a partir de requisições reais, validando a eficácia do gerenciador de requisições.

Os testes foram realizados em uma máquina onde o gerenciador de requisições foi avaliado em condições reais de uso. Adicionalmente, testes sob as mesmas condições foram conduzidos utilizando o *tpm-tools*, uma ferramenta amplamente utilizada para interações com TPM. Os resultados comparativos mostram um desempenho muito similar entre as duas abordagens. Tanto o gerenciador desenvolvido com a biblioteca *go-tpm* quanto o *tpm-tools* demonstraram eficiência e eficácia na gestão das requisições ao TPM, confirmando a robustez e a competitividade da implementação proposta.

O objetivo da implementação é demonstrar que, independentemente do número de requisições feitas ao TPM, o gerenciador será capaz de atender todas as requisições sem que o TPM trave. Levando isso em consideração, outro teste realizado foi utilizando o gerenciador e sem utilizar o gerenciador. O código de ambas implementações faz uso da biblioteca Go-TPM.

A Figura 1 ilustra que as execuções realizadas sem o gerenciador são mais rápidas do que aquelas utilizando a solução proposta. O teste, conduzido com diferentes tempos de espera e uma carga fixa de 900 requisições, revelou que as execuções sem o gerenciador foram, em média, 10 segundos mais rápidas. Contudo, essa maior velocidade vem com um custo: a ausência de tempos de espera resulta em erros no chip. Sem o gerenciador, as requisições não são atendidas, pois o TPM trava sem os intervalos necessários. Com o gerenciador, no entanto, as requisições são processadas, mesmo sem tempos de espera. A inclusão dos tempos de espera na solução com o gerenciador garante que os testes sejam conduzidos em condições equivalentes.

Algumas barras não foram representadas na figura 1 devido aos erros que o código retornou quando o gerenciador não foi utilizado. Nos tempos de 0,006 segundos, 0,003 segundos e 0 segundos o código retorna um erro explicando que não foi possível executar as operações no TPM. Esse erro se refere às múltiplas requisições que o TPM recebeu, mas em seguida travou. O tempo mínimo para que o TPM não retornasse um erro foi 0,009 segundos. Em comparativo, tem-se que o gerenciador realiza as devidas operações e o TPM não trava - o tempo médio de execução das 900 requisições e sem tempo de espera foi de 4,83 segundos.

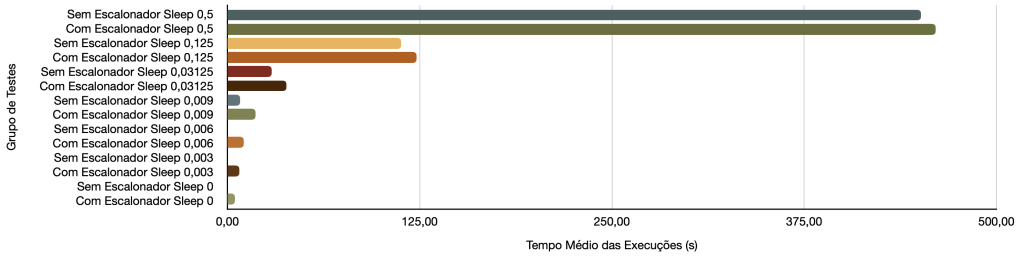

Figura 1. Tempos de Execução

Na Figura 2, tem-se a quantidade de requisições atendidas. Nesse teste, usando o gerenciador, todas as requisições foram atendidas, assim como até o tempo de espera de 0,009 segundos na implementação sem utilizar gerenciador. No tempo de 0,006 segundos, em algumas execuções as requisições não foram atendidas ou apenas parcialmente atendidas; e nos tempos de 0,003 segundos e 0 segundos, nenhuma requisição foi atendida sem a utilização do gerenciador.

Além dos testes realizados em uma máquina física, foram conduzidos testes em máquinas virtuais para confirmar que a implementação proposta também funcionaria em um ambiente virtual. Esses testes adicionais garantiram que o gerenciador de requisições mantivesse a mesma eficiência e eficácia ao lidar com chamadas concorrentes ao TPM em um ambiente virtualizado. Os resultados foram igualmente positivos, demonstrando que a implementação é versátil, funcionando corretamente tanto em hardware físico quanto em ambientes virtuais.

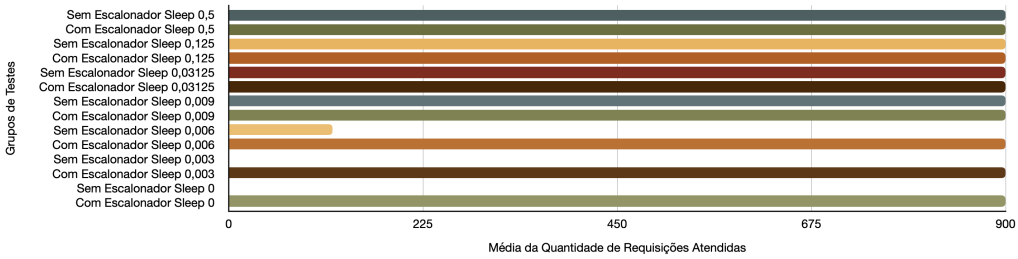

Figura 2. Quantidade de Requisições Atendidas

6. Conclusão e Trabalhos Futuros

Neste trabalho, foi explorado a implementação de um gerenciador de requisições em Golang para o TPM, visando melhorar a segurança em ambientes virtualizados. Através de testes em máquinas físicas e virtuais, foi validado a eficiência e a eficácia da solução, que se mostrou robusta e capaz de manter a integridade dos dados mesmo sob alta demanda de requisições simultâneas.

A avaliação do gerenciador de requisições ao TPM foi realizada com um conjunto fixo de 900 requisições, variando os tempos de espera entre 0,5 segundos, 0,125 segundos, 0,03125 segundos, 0,009 segundos, 0,006 segundos e 0,003 segundos. Observou-se que, com tempos de espera menores, a taxa de requisição aumentou significativamente. No entanto, a taxa de atendimento caiu drasticamente nesse cenário, com nenhuma das requisições sendo atendida devido ao travamento do TPM na implementação onde o gerenciador não é utilizado. Em contraste, com o uso do gerenciador, a taxa de atendimento manteve-se alta, com todas as 900 requisições sendo atendidas eficientemente em um tempo médio de 4,83 segundos. Esse resultado demonstra que o gerenciador não apenas previne o travamento do TPM, mas também garante uma alta taxa de atendimento, mesmo sob cargas elevadas de requisições. A implementação do gerenciador está disponível no GitHub: `https://github.com/marianezei/ArtigoWTICGSBSeg24`.

Além disso, a implementação foi comparada com a ferramenta *tpm-tools*, demonstrando que o gerenciador oferece um desempenho similar, reforçando sua competitividade e viabilidade. Espera-se que as descobertas e implementações sugeridas sirvam como uma base sólida para futuras pesquisas e desenvolvimentos na área de segurança.

Como trabalhos futuros, sugere-se a exploração de diferentes algoritmos de escalonamento que possam complementar a solução desta pesquisa, como, por exemplo, algoritmos que considerem a prioridade de máquinas virtuais. Além disso, podem ser investigados algoritmos adaptativos que ajustem dinamicamente o escalonamento baseado na carga do sistema e no comportamento histórico das requisições.

Referências

[1] W. Arthur, D. Challener, and K. Goldman, *A practical guide to TPM 2.0: Using the new trusted platform module in the new age of security*. Springer Nature, 2015.

[2] M. Zimmerman. (2018) Virtual trusted platform module for shielded vms: security in plaintext. Acessado em 17 de maio de 2024. [Online]. Available: https://cloud.google.com/blog/products/identity-security/virtual-trusted-platform-module-for-shielded-vms-security-in-plaintext

[3] M. Eckel and T. Riemann, "Userspace software integrity measurement," in *Proceedings of the 16th International Conference on Availability, Reliability and Security*, 2021, pp. 1–11.

[4] Trusted computing group. Acessado em 17 de maio de 2024. [Online]. Available: https://trustedcomputinggroup.org/trusted-computing

[5] M. Tassyany, R. Sarmento, E. Falcão, R. Gomes, and A. Brito, "Um mecanismo de aprovisionamento de identidades para microsserviços baseado na integridade do ambiente de execução," in *Anais do XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. SBC, 2021, pp. 714–727.

[6] R. Perez, R. Sailer, L. van Doorn *et al.*, "vtpm: virtualizing the trusted platform module," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 305–320.

[7] J. Schmitz, J. Loew, J. Elwell, D. Ponomarev, and N. Abu-Ghazaleh, "Tpm-sim: A framework for performance evaluation of trusted platform modules," in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 236–241.

[8] Trusted virtual data center. Acessado em 17 de maio de 2024. [Online]. Available: http://domino.research.ibm.com/comm/research\%20projects.nsf/pages/ssd\%20trustedvirtualdatacenter.index.html