Simpósio de Excelência em Gestão e Tecnologia
XXI SEGeT
GOVERNANÇA, RESPONSABILIDADE SOCIAL E O MEIO AMBIENTE
Desafios para uma Gestão Social, Financeira e Sustentabilidade Empresarial
Realização: UniDomBosco Centro Universitário Dom Bosco do Rio de Janeiro unesp
Patrocínio: INB

UMA SIMULAÇÃO DE ATAQUE QUÂNTICO AO SPHINCS+

Bruno Leonardo Santos Menezes
bruno.menezes@prof.ceteprsd.faetec.rj.gov.br
Faetec Resende - RJ

Franklin de Lima Marquezino
franklin@cos.ufrj.br
UFRJ

Claudio Miceli de Farias
cmicelifarias@cos.ufrj.br
UFRJ

Resumo: Este artigo tem como objetivo avaliar a resistência e integridade do algoritmo SPHINCS+ no ambiente da criptografia pós-quântica. SPHINCS+ faz uso de uma estrutura de hypertree de alta complexidade e assinatura eXtended Merkle Signature Scheme em camadas. Além de Forest of Random Subsets para assinatura de mensagens. SPHINCS+ pode resistir de forma segura a ataques de computadores quânticos, sendo capaz de garantir a integridade dos dados. Os experimentos de geração de chaves, assinatura e verificação, integração dos componentes, casos extremos e limites, entradas aleatórias, respostas conhecidas e resistência quântica em ambiente simulado foram satisfatórios. Este estudo pode representar uma contribuição relevante, como uma proposta metodológica para avaliar a resistência neste ambiente, contribuindo para que sejam realizadas futuras pesquisas em ambiente simulado.

Palavras Chave: Segurança informação - computação quântica - criptografia -pós-quântica - ataque quântico

1. INTRODUÇÃO

A constante evolução da computação quântica, ressalta a importância de algoritmos que sejam capazes de resistir a ataques oriundos de computadores quânticos. De forma geral, algoritmos criptográficos tradicionais são vulneráveis a ataques quânticos. Como consequência, todos os sistemas que utilizam estes modelos correm risco de estarem inseguros.

Este artigo tem como objetivo avaliar a resistência e integridade do algoritmo *SPHINCS+* no ambiente da criptografia pós-quântica. *SPHINCS+* merece destaque, como um dos principais algoritmos que são capazes de resistir a ataques quânticos.

Foi realizado levantamento bibliográfico em relação ao tema, investigando o estado da arte envolvendo *SPHINCS+*. Existe uma carência de estudos científicos em relação ao tema. Os experimentos com *SPHINCS+* foram implementados com a utilização das linguagens de programação *C* e *Python*.

Foram realizados testes de geração de chaves, assinatura e verificação, integração dos componentes, casos extremos e limites, entradas aleatórias, respostas conhecidas e resistência quântica em ambiente simulado. A realização da simulação de ataque quântico pode representar uma proposta metodológica relevante, possibilitando que sejam realizados outros estudos sobre o assunto.

2. DESENVOLVIMENTO

Para Bernstein et al. (2015) os desafios fundamentais para a criptografia no contexto pós-quântico, consiste na capacidade e facilidade que computadores quânticos podem quebrar sistemas clássicos, tornando os modelos tradicionais inseguros. O *SPHINCS* representou avanços e apresentou vantagens neste contexto.

Com a evolução da criptografia neste cenário, emerge *SPHINCS+*, uma evolução do *SPHINCS*. Bernstein et al. (2019) apresenta *SPHINCS+* como um *framework* de assinatura, ou seja, um esquema de assinatura *hash* sem estado, projetado como uma opção no ambiente da criptografia pós-quântica. O *SPHINCS+* é uma generalização do *SPHINCS*, foi criado com objetivo de otimizar a velocidade, tamanho da assinatura e a segurança de forma geral. Em 2022 o *National Institute of Standards and Technology* (NIST), instituição vinculada ao *U.S. Department of Commerce*, com a missão, em seu "*site* oficial": promover a inovação e a competitividade industrial dos Estados Unidos da América através do avanço da ciência, dos padrões e da tecnologia de medição de forma a aumentar a segurança econômica e melhorar a nossa qualidade de vida. (NIST, 2024).

O NIST conduziu o *post-quantum cryptography standardization project*, com objetivo iniciar o desenvolvimento de soluções na transição dos atuais algoritmos de criptografia e chave pública para rotinas resistentes a ataques baseados em computadores quânticos. O *SPHINCS+*, obteve destaque, apesar de ser considerado maior e mais lento que os outros também selecionados, *SPHINCS+* foi considerado útil como *backup*, pois baseia-se em funções *hash*, que para Kelsey e Schneier (2005) são algoritmos que transformam uma entrada de tamanho variável em uma saída de tamanho fixo.

Para Bernstein e Hülsing (2019) *SPHINCS+* aproveita os aspectos de segurança como resistência à pré-imagem e segunda pré-imagem (do inglês *preimage resistance* - PRE e *second-preimage resistance* - SPR, respectivamente). PRE de uma função *hash* H pode ser definida como a dificuldade de encontrar qualquer entrada x' tal que H(x') seja igual a um valor de hash dado y, formalizada pela probabilidade de sucesso de um algoritmo A, ou seja: $\mathrm{SuccHpre}(A)=\Pr[x \leftarrow RX; x' \leftarrow A(H(x)) : H(x)=H(x')]$. SPR, pode ser definido como a

dificuldade de encontrar uma entrada diferente x' para uma dada entrada x tal que H(x') = H(x), SuccHsprH(A)=Pr[x←RX;x′←A(x):H(x)=H(x′)∧x≠x′].

A resistência à segunda pré-imagem (do inglês *decisional second-preimage resistance* - DSPR), pode ser entendida como a vantagem negligenciável de um atacante em decidir se uma entrada x tem uma segunda pré-imagem, com a vantagem DSPR definida como AdvHdspr(A)=max{0,Pr[x←RX;b←A(x):P(x)=b]−p} onde P(x)=1P(x) se x tem uma PRE e p trata-se probabilidade de que uma entrada aleatória tenha uma DSPR. A combinação de SPR e DSPR implica PRE, fundamental para a segurança de *SPHINCS+*, já que depende de funções *hash* seguras para obter integridade e autenticidade.

De acordo com Jean-Philippe Aumasson, Daniel J et al. (2022) *SPHINCS+* é um sistema de assinatura digital pós-quântica baseado em *hash* e árvore de *Merkle* que não guarda estado. Um aperfeiçoamento do *SPHINCS*. *SPHINCS+* faz uso de uma estrutura de hypertree de alta complexidade e assinatura *eXtended Merkle Signature Scheme* (XMSS) em camadas. Além de *Forest of Random Subsets* (FORS) para assinatura de mensagens.

Os resultados obtidos por Singh et al. (2023) na tabela 1 indicam que todas as funções individuais passaram nos testes de unidade, enquanto a integração entre componentes foi validada de forma satisfatória.

Tabela 1: Resultados dos testes para SPHINCS+.

Tipo de Teste	Resultado Resumido
Teste de Unidade	Funções passaram nos testes com diversas entradas.
Teste de Integração	Interação entre componentes validados.
Teste de Casos Extremos e Limites	Efetivo com entradas extremas e nos limites.
Teste Aleatório	Testes com aleatoriedade não revelaram problemas.
Teste com Respostas Conhecidas	Saídas de acordo com os vetores de teste.
Teste de Desempenho	Eficiente em tempo e uso de recursos.
Análise de Segurança	Resistente a ataques comuns, sem fraquezas.
Teste Comparativo	Competitivo em desempenho e segurança.
Resistência a Colisões	Previne colisões de hash.
Avaliação de Resistência Quântica	Robusto contra ataques quânticos.
Eficiência e Uso de Recursos	Eficiente sob diferentes cargas de trabalho.
Análise de Força de Segurança e Vulnerabilidades	Resiste a ataques criptográficos, sem vulnerabilidades importantes.

Fonte: Adaptado de Singh et al (2023)

SPHINCS+ comportou-se de forma efetiva com entradas extremas e aleatórias. É competitivo em desempenho e segurança, eficaz em prevenir colisões de hash e resistente a ameaças de computadores quânticos e ataques criptográficos. Eficiência confirmada no tempo

de execução quanto e na utilização de recursos, considerando inclusive diferentes cargas de trabalho, não ocorrem vulnerabilidades relevantes.

De acordo com Marel et al. (2023) a realidade da computação quântica, aumentou os desafios da criptografia tradicional, algoritmos clássicos podem ser quebrados por computadores quânticos. Neste contexto, atualmente, o estado da arte em criptografia pós-quântica passa pelo desenvolvimento e padronização de novos algoritmos capazes de resistir a ataques quânticos. São quatro algoritmos principais: *CRYSTALS-Kyber*, *Classic McEliece*, *SPHINCS+* e *TFQKD*. O investimento contínuo, por parte do NIST, no desenvolvimento e na padronização desses códigos, revelam um avanço contra ataques oriundos da computação quântica. Vale ressaltar, existem algumas lacunas significativas neste campo de estudo: a transmissão de partículas emaranhadas a longas distâncias ainda é um problema desafiador, além da confiabilidade dos *qubits*. Este tipo de criptografia (quântica) necessita de equipamentos específicos, podendo aumentar os custos de implementação. Existem também imperfeições na geração e medição de fótons, protocolos de distribuição de chaves quânticas também apresentam vulnerabilidades.

O desenvolvimento de algoritmos seguros, neste contexto, é um desafio contínuo. Existindo diversas possibilidades de implementações de soluções em larga escala. Analisando os estudos de Buchmann et al. (2016) e Marel et al. (2023), *SPHINCS+* apresenta vantagens na realidade da criptografia pós-quântica, funções hash podem proporcionar segurança contra ataques quânticos, com PRE e SPR, fundamentais para obter integridade e autenticidade das assinaturas digitais. A estrutura de hiper-árvore e o XMSS, FORS aumentam a segurança e a eficiência. Experimentos atuais, atestaram a eficácia de *SPHINCS+* em diversas condições, resultando em um desempenho significativo e resistência comprovada. Existem limitações, uma delas é alcançar desempenho e segurança com o *SPHINCS+*, apresentando assinaturas maiores e menor velocidade em comparação aos demais sistemas pós-quânticos. Em sistemas com recursos limitados, isto é desafiador. A decisão referente a alguns parâmetros: *Winternitz* (w) e o comprimento do *hash* (len_1), é fundamental neste contexto envolvendo segurança e desempenho. A escolha desses parâmetros de forma inadequada pode reduzir a eficiência e como consequência a segurança. Para implementar *SPHINCS+* faz-se necessário gerenciar grandes estruturas de dados e múltiplas operações de hash, isto pode gerar vulnerabilidades. A compatibilidade com infraestruturas existentes é outro desafio. Esses aspectos revelam carência de estudos que tratam do tema, existindo oportunidades de pesquisas ainda inexploradas neste campo de estudo.

3. METODOLOGIA

O estudo realizado por Singh et al. (2023) fornece possibilidades de realizar testes com o algoritmo objeto deste estudo, seguindo este direcionamento, foi possível clonar o repositório do projeto oficial NIST *post-quantum crypto project* (https://github.com/sphincs/sphincsplus.git) em uma máquina local. Foram definidos os parâmetros no arquivo *MakeFile* (https://github.com/sphincs/sphincsplus/blob/master/ref/Makefile) *PARAMS=sphincs-shake-256f* e *THASH=robust*. O compilador foi o *GCC (CC=/usr/bin/gcc)*, e as *flags* de compilação (CFLAGS) incluem *-Wall*, *-Wextra* e *-Wpedantic* para verificação de erros, *-O0* para desativar otimizações, *-std=c99* para aderir ao padrão *C99*, e parâmetros de integração com *python* (*-I/usr/include/python3.12 -L/usr/lib/python3.12/config-3.12-x86_64-linux-gnu*). As bibliotecas de linkagem (LDLIBS) foram *cmocka* para testes unitários, crypto para funções criptográficas, e *python 3.12* para scripts python. As fontes e cabeçalhos fundamentais foram especificadas, com regras condicionais (ifneq) ajustando os arquivos com base no algoritmo de *hash* (*shake*, *haraka*,

sha2) definido em *PARAMS*. A compilação abrange alvos como *PQCgenKAT_sign*, *tests*, *benchmarks* e *clean*, permitindo a compilação do binário, testes unitários e de integração, além de *benchmarks* de desempenho. Cada teste foi configurado para compilar com as fontes e cabeçalhos adequados, os executáveis são gerados de forma automática. A limpeza (clean) remove todos os arquivos gerados. Para a realização dos experimentos foi utilizado um computador com arquitetura x86_64, operando com o *Fedora* 40, *kernel* versão *6.9.7-200.fc40.x86_64*. Processador *Intel(R) Core(TM) i5-10400 CPU* com 6 núcleos, 12 *threads*, frequência base de 2.90 *GHz* e máxima de 4.30 *GHz*, suportando instruções de 32 e 64 *bits*. 16 *GB* de memória *RAM*, configuração de *cache* distribuída em 192 *KiB* para *L1d* e *L1i*, 1.5 *MiB* para *L2* e 12 *MiB* para *L3*. *SSD* de 476.9 *GB* para armazenamento principal e um disco adicional de 465.8 *GB*. Virtualização é suportada por *VT-x Intel*. *Intel UHD Graphics* 630 como controladora gráfica integrada.

Este estudo desenvolveu codificação na linguagem de programação *C* e *Python* (anexo I e II), utilizando a biblioteca *CMocka* (https://cmocka.org/), possibilitando a realização de experimentos com *SPHINCS+*, com base no estudo realizado por Singh et al. (2023), foram realizados com objetivo de validar em ambiente simulado a robustez e a integridade do *SPHINCS+* no ambiente pós-quântico.

Conforme anexo I e II, o teste de geração de chaves (*test_keygen*) verificou a capacidade em criar chaves públicas e privadas, verificando a correta inicialização e armazenamento. A função *test_sign_verify* validou a funcionalidade ao assinar uma mensagem e verificar a assinatura, assegurando que a mensagem original pudesse ser autenticada, *test_integration* testou a integração dos componentes, combinando os processos de geração de chaves, assinatura e verificação, atestando a coerência e a correta interação entre os módulos. A verificação de casos extremos (*test_edge_cases*) avaliou o desempenho com mensagens de tamanhos variados, curtas e longas, validando a integridade em situações extremas. O teste de aleatoriedade (*test_randomized*) realizou múltiplas iterações de assinatura e verificação com mensagens geradas aleatoriamente, atestando a consistência sob diferentes condições de entrada. O teste com respostas conhecidas (*test_known_answers*) utilizou mensagens e assinaturas predefinidas para validar a precisão sob condições controladas, enquanto o teste de resistência quântica (*test_quantum_resistance*) foi inovador, pois simulou um ataque quântico, utilizando um *script python* que utiliza a simulação de um circuito quântico através da biblioteca *Qiskit* (https://docs.quantum.ibm.com/api/qiskit/circuit_library), avaliando a resiliência *SPHINCS+*, contra uma ameaça quântica. Este último teste é fundamental para demonstrar a eficácia na segurança das assinaturas digitais, diante de ataques de computadores quânticos. O código em questão ilustra a criação e simulação de um circuito quântico utilizando a biblioteca *Qiskit*, com foco na implementação de um ataque quântico que explora as propriedades fundamentais do entrelaçamento e da superposição quântica.

Conforme anexo II, O circuito inicializa dois *qubits* e dois *bits* clássicos, seguidos pela aplicação de uma porta *Hadamard* (H) no primeiro *qubit*, H adiciona o primeiro qubit em uma superposição igual de estados $|0\rangle$ e $|1\rangle$, criando de entrelaçamento. Uma porta CNOT (*controle-NOT*) é aplicada entre o primeiro *qubit* (controle) e o segundo (alvo). Entrelaçando os qubits, assim o estado do segundo *qubit* depende do primeiro, ou seja, um estado *Bel*. Ambos qubits são medidos, seus resultados são armazenados nos *bits*. A transpilação (processo de converter um circuito quântico, escrito em de alto nível, em uma sequência equivalente de operações que podem ser executadas por um simulador) do circuito melhora a sequência de operações do simulador *qasm_simulator*. A execução gera resultados que são contados e exibidos, possibilitando análise das distribuições de probabilidade dos estados

finais dos qubits. As funções atacadas (anexo I) são *crypto_sign_keypair*, *crypto_sign* e *crypto_sign_open*, essenciais para garantir a segurança do *SPHINCS+*.

4. RESULTADOS

Os experimentos foram conduzidos para avaliar a resistência e integridade do SPHINCS+. Os resultados indicam desempenho consistente e satisfatório (tabela 2).

Tabela 2: Experimentos com *SPHINCS+*.

Testes	Resultado
test_keygen	Sucesso
test_sign_verify	Sucesso
test_integration	Sucesso
test_edge_cases	Sucesso
test_randomized	Sucesso
test_known_answers	Sucesso
test_quantum_resistance	Sucesso

Fonte: Autores (2024)

O teste de geração de chaves foi executado com sucesso, validando a capacidade do algoritmo em criar chaves públicas e privadas, teste de assinatura e verificação confirmou que a mensagem original foi assinada e verificada também com sucesso, resultando em mensagem equivalente à original. O teste de integração combinou a geração de chaves, assinatura e verificação, ou seja, existe coerência e interação entre módulos. O teste de casos extremos foi realizado com sucesso. As interações do teste de aleatoriedade foram concluídas com sucesso, existe consistência com diferentes entradas. No teste com respostas conhecidas, também obteve sucesso. O teste de resistência quântica apresentou a distribuição de probabilidade ('11': 497, '00': 527), indicando chaves sendo geradas e assinaturas verificadas com sucesso. Neste contexto, indica que o estado 11 (ambos os qubits medidos no estado 1) foi observado 497 vezes, no estado 00 (também avaliados no estado 0) ocorreu 527 vezes. Essa proximidade de contagens apresenta uma distribuição equilibrada, com a simulação do circuito quântico foi realizada com sucesso.

5. CONSIDERAÇÕES FINAIS

Este estudo atestou a eficiência e eficácia do algoritmo *SPHINCS+* no ambiente da criptografia pós-quântica. Com experimentos realizados, em ambiente simulado, foi possível validar a segurança do algoritmo em diferentes condições. Os resultados demonstraram que o mesmo tem capacidade de gerar e verificar chaves de forma precisa, garantindo a integridade das assinaturas digitais, mesmo após ocorrerem simulações de ataques de computadores quânticos. Estes aspectos confirmam a importância do *SPHINCS+*, como uma das possíveis soluções para a criptografia e os desafios da área com o desenvolvimento e constante evolução da computação quântica.

A estrutura de *hash* e a utilização de árvores de *Merkle* possibilitam resistir a ataques. Porém, ainda existem possibilidades de otimização, principalmente na velocidade e na redução do tamanho das assinaturas.

O desenvolvimento de um *script Python*, que realiza simulação de ataques oriundos de computadores quânticos, com *Qiskit*, pode representar uma contribuição relevante, fornecendo um método para avaliar a resistência neste ambiente. Contribuindo para que sejam realizadas pesquisas futuras e melhorias, com o desenvolvimento, em ambiente simulado, de ataques quânticos ao *SPHINCS+* com maior complexidade.

6. REFERÊNCIAS

ALVARADO, MAREL ET AL. A Survey on Post-Quantum Cryptography: State-of-the-Art and Challenges. arXiv preprint arXiv:2312.10430, 2023. Disponível em: https://arxiv.org/pdf/2312.10430v1.pdf. Acesso em: 01 abril. 2024.

BERNSTEIN, DANIEL J. ET AL. SPHINCS: practical stateless hash-based signatures. In: Annual international conference on the theory and applications of cryptographic techniques. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. p. 368-397. Disponível em: https://link.springer.com/chapter/10.1007/978-3-662-46800-5_15. Acesso em: 01 mar. 2024.

BERNSTEIN, DANIEL J. ET AL. The SPHINCS+ signature framework. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security. 2019. p. 2129-2146. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3319535.3363229. Acesso em: 01 mar. 2024.

BERNSTEIN, DANIEL J.; HÜLSING, ANDREAS. Decisional second-preimage resistance: When does SPR imply PRE?. In: Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part III 25. Springer International Publishing, 2019. p. 33-62. Disponível em: https://link.springer.com/chapter/10.1007/978-3-030-34618-8_2. Acesso em: 01 mar. 2024.

BUCHMANN, JOHANNES A. ET AL. Post-quantum cryptography: state of the art. The New Codebreakers: Essays Dedicated to David Kahn on the Occasion of His 85th Birthday, p. 88-108, 2016. Disponível em: https://www.academia.edu/download/107103556/article.pdf. Acesso em: 01 abril. 2024.

JEAN-PHILIPPE AUMASSON, DANIEL J. ET AL. SPHINCS+. Submission to the NIST post-quantum project. v.3.1, June 10, 2022. Disponível em: https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms. Acesso em: 01 mar. 2024.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Site Oficial, 2024. Disponível em: https://www.nist.gov/about-nist. Acesso em: 01 mar. 2024

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Site Oficial, 2022. Disponível em: https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms. Acesso em: 01 mar. 2024.

SINGH, SONIA. KARTHICK, SRAVAN. PRADA, SHUBHAPRADA. Investigating SHA and Proposing SPHINCS+ as a Post Quantum Algorithm (PQC), International Journal for Research in Applied Science & Engineering Technology, Vol. 11, Issue IX, IJRASET, 2023. Disponível em: https://doi.org/10.22214/ijraset.2023.55872. Acesso em: 30 maio 2024.

SINGH, HARSHDEEP. Code based cryptography: Classic mceliece. arXiv preprint arXiv:1907.12754, 2019. Disponível em: https://arxiv.org/pdf/1907.12754. Acesso em: 01 abril. 2024.

KELSEY J. E SCHNEIER B. Second Preimages on n-bit Hash Functions for Much Less than 2n Work, Lecture Notes in Computer Science, Vol. 3494, Springer, 2005. Disponível em: https://link.springer.com/chapter/10.1007/11426639_28. Acesso em: 01 mar. 2024.

ANEXO I

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include "../ref/api.h"
#include "../ref/params.h"
#include "../ref/randombytes.h"
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <openssl/sha.h>
#include <Python.h>
#define MESSAGE_LEN 32
#define SIGNATURE_LEN CRYPTO_BYTES
#define PUBLIC_KEY_LEN CRYPTO_PUBLICKEYBYTES
#define SECRET_KEY_LEN CRYPTO_SECRETKEYBYTES
static void print_hex(const unsigned char *data, size_t len) {
        for (size_t i = 0; i < len; ++i) {
        printf("%02x ", data[i]);
        }
        printf("\n");
}
static void test_keygen(void **state) {
        unsigned char pk[PUBLIC_KEY_LEN];
        unsigned char sk[SECRET_KEY_LEN];
        memset(pk, 0, PUBLIC_KEY_LEN);
        memset(sk, 0, SECRET_KEY_LEN);
        int result = crypto_sign_keypair(pk, sk);
        printf("Key Generation Result: %d\n", result);
        printf("Public Key: ");
        print_hex(pk, PUBLIC_KEY_LEN);
        printf("Secret Key: ");
        print_hex(sk, SECRET_KEY_LEN);

        assert_int_equal(result, 0);
        assert_non_null(pk);
        assert_non_null(sk);
}
static void test_sign_verify(void **state) {
        unsigned char message[MESSAGE_LEN] = {0};
        unsigned char signature[SIGNATURE_LEN];
        unsigned char pk[PUBLIC_KEY_LEN];
        unsigned char sk[SECRET_KEY_LEN];
```

```
        unsigned long long siglen = 0;
        memset(signature, 0, SIGNATURE_LEN);
        memset(pk, 0, PUBLIC_KEY_LEN);
        memset(sk, 0, SECRET_KEY_LEN);
        int keygen_result = crypto_sign_keypair(pk, sk);
        printf("Key Generation Result: %d\n", keygen_result);
        assert_int_equal(keygen_result, 0);
        int sign_result = crypto_sign(signature, &siglen, message, MESSAGE_LEN, sk);
        printf("Sign Result: %d\n", sign_result);
        printf("Signature: ");
        print_hex(signature, siglen);
        assert_int_equal(sign_result, 0);
        unsigned long long mlen = MESSAGE_LEN;
        unsigned char verified_message[MESSAGE_LEN];
        memset(verified_message, 0, MESSAGE_LEN);
        int verify_result = crypto_sign_open(verified_message, &mlen, signature, siglen, pk);
        printf("Verify Result: %d\n", verify_result);
        printf("Verified Message: ");
        print_hex(verified_message, mlen);
        assert_int_equal(verify_result, 0);
        assert_memory_equal(message, verified_message, MESSAGE_LEN);
}
static void test_integration(void **state) {
        unsigned char message[MESSAGE_LEN] = {0};
        unsigned char signature[SIGNATURE_LEN];
        unsigned char pk[PUBLIC_KEY_LEN];
        unsigned char sk[SECRET_KEY_LEN];
        unsigned long long siglen = 0;
        memset(signature, 0, SIGNATURE_LEN);
        memset(pk, 0, PUBLIC_KEY_LEN);
        memset(sk, 0, SECRET_KEY_LEN);
        int keygen_result = crypto_sign_keypair(pk, sk);
        printf("Key Generation Result: %d\n", keygen_result);
        assert_int_equal(keygen_result, 0);
        int sign_result = crypto_sign(signature, &siglen, message, MESSAGE_LEN, sk);
        printf("Sign Result: %d\n", sign_result);
        printf("Signature: ");
        print_hex(signature, siglen);
        assert_int_equal(sign_result, 0);
        unsigned long long mlen = MESSAGE_LEN;
        unsigned char verified_message[MESSAGE_LEN];
        memset(verified_message, 0, MESSAGE_LEN);
        int verify_result = crypto_sign_open(verified_message, &mlen, signature, siglen, pk);
        printf("Verify Result: %d\n", verify_result);
```

```
        printf("Verified Message: ");
        print_hex(verified_message, mlen);
        assert_int_equal(verify_result, 0);
        assert_memory_equal(message, verified_message, MESSAGE_LEN);
}
static void test_edge_cases(void **state) {
        unsigned char short_message[1] = {0};
        unsigned char long_message[1024] = {0}; // ajustável conforme necessário
        unsigned char signature[SIGNATURE_LEN];
        unsigned char pk[PUBLIC_KEY_LEN];
        unsigned char sk[SECRET_KEY_LEN];
        unsigned long long siglen = 0;
        memset(signature, 0, SIGNATURE_LEN);
        memset(pk, 0, PUBLIC_KEY_LEN);
        memset(sk, 0, SECRET_KEY_LEN);
        int keygen_result = crypto_sign_keypair(pk, sk);
        printf("Key Generation Result: %d\n", keygen_result);
        assert_int_equal(keygen_result, 0);
        int sign_result = crypto_sign(signature, &siglen, short_message, 1, sk);
        printf("Sign Result for Short Message: %d\n", sign_result);
        printf("Signature: ");
        print_hex(signature, siglen);
        assert_int_equal(sign_result, 0);
        unsigned long long mlen = 1;
        unsigned char verified_message[1] = {0};
        int verify_result = crypto_sign_open(verified_message, &mlen, signature, siglen, pk);
        printf("Verify Result for Short Message: %d\n", verify_result);
        printf("Verified Short Message: ");
        print_hex(verified_message, mlen);
        assert_int_equal(verify_result, 0);
        assert_memory_equal(short_message, verified_message, 1);
        sign_result = crypto_sign(signature, &siglen, long_message, 1024, sk);
        printf("Sign Result for Long Message: %d\n", sign_result);
        printf("Signature: ");
        print_hex(signature, siglen);
        assert_int_equal(sign_result, 0);
        mlen = 1024;
        unsigned char verified_long_message[1024] = {0};
        verify_result = crypto_sign_open(verified_long_message, &mlen, signature, siglen, pk);
        printf("Verify Result for Long Message: %d\n", verify_result);
        printf("Verified Long Message: ");
        print_hex(verified_long_message, mlen);
        assert_int_equal(verify_result, 0);
        assert_memory_equal(long_message, verified_long_message, 1024);
```

```
}
static void test_randomized(void **state) {
        unsigned char message[MESSAGE_LEN];
        unsigned char signature[SIGNATURE_LEN];
        unsigned char pk[PUBLIC_KEY_LEN];
        unsigned char sk[SECRET_KEY_LEN];
        unsigned long long siglen = 0;
        memset(signature, 0, SIGNATURE_LEN);
        memset(pk, 0, PUBLIC_KEY_LEN);
        memset(sk, 0, SECRET_KEY_LEN);
        for (int i = 0; i < 100; i++) {
        randombytes(message, MESSAGE_LEN);
        int keygen_result = crypto_sign_keypair(pk, sk);
        printf("Key Generation Result: %d\n", keygen_result);
        assert_int_equal(keygen_result, 0);
        int sign_result = crypto_sign(signature, &siglen, message, MESSAGE_LEN, sk);
        printf("Sign Result: %d\n", sign_result);
        printf("Signature: ");
        print_hex(signature, siglen);
        assert_int_equal(sign_result, 0);
        unsigned long long mlen = MESSAGE_LEN;
        unsigned char verified_message[MESSAGE_LEN];
        memset(verified_message, 0, MESSAGE_LEN);
        int verify_result = crypto_sign_open(verified_message, &mlen, signature, siglen, pk);
        printf("Verify Result: %d\n", verify_result);
        printf("Verified Message: ");
        print_hex(verified_message, mlen);
        assert_int_equal(verify_result, 0);
        assert_memory_equal(message, verified_message, MESSAGE_LEN);
        }
}
static void test_known_answers(void **state) {
        unsigned char known_message[MESSAGE_LEN] = "Hello World!           "; // Ajustar para 32 bytes
        unsigned char signature[SIGNATURE_LEN];
        unsigned char pk[PUBLIC_KEY_LEN];
        unsigned char sk[SECRET_KEY_LEN];
        unsigned long long siglen = 0;
        memset(signature, 0, SIGNATURE_LEN);
        memset(pk, 0, PUBLIC_KEY_LEN);
        memset(sk, 0, SECRET_KEY_LEN);
        int keygen_result = crypto_sign_keypair(pk, sk);
        printf("Key Generation Result: %d\n", keygen_result);
        assert_int_equal(keygen_result, 0);
        int sign_result = crypto_sign(signature, &siglen, known_message, MESSAGE_LEN, sk);
```

```
        printf("Sign Result: %d\n", sign_result);
        printf("Signature: ");
        print_hex(signature, siglen);
        assert_int_equal(sign_result, 0);
        unsigned long long mlen = MESSAGE_LEN;
        unsigned char verified_message[MESSAGE_LEN];
        memset(verified_message, 0, MESSAGE_LEN);

        printf("Known Message: ");
        print_hex(known_message, MESSAGE_LEN);
        printf("Generated Signature: ");
        print_hex(signature, siglen);
        printf("Public Key: ");
        print_hex(pk, PUBLIC_KEY_LEN);
        int result = crypto_sign_open(verified_message, &mlen, signature, siglen, pk);
        printf("Verification Result: %d\n", result);
        if (result != 0) {
        printf("Failed Verification Message: ");
        print_hex(verified_message, mlen);
        }
        assert_int_equal(result, 0);
        assert_memory_equal(known_message, verified_message, MESSAGE_LEN);
}
static void test_quantum_resistance(void **state) {
        printf("Inicializando o Python...\n");
        Py_Initialize();
        if (!Py_IsInitialized()) {
        printf("Failed to initialize Python interpreter\n");
        return;
        }
        printf("Alocando memória...\n");
        unsigned char pk[PUBLIC_KEY_LEN];
        unsigned char sk[SECRET_KEY_LEN];
        unsigned char message[MESSAGE_LEN] = "Test message for SPHINCS+";
        unsigned char signature[SIGNATURE_LEN];
        unsigned long long siglen = 0;
        unsigned long long mlen = MESSAGE_LEN;
        unsigned char verified_message[MESSAGE_LEN];
        printf("Inicializando variáveis...\n");
        memset(pk, 0, PUBLIC_KEY_LEN);
        memset(sk, 0, SECRET_KEY_LEN);
        memset(signature, 0, SIGNATURE_LEN);
        memset(verified_message, 0, MESSAGE_LEN);
        printf("Gerando chaves...\n");
```

```c
int keygen_result = crypto_sign_keypair(pk, sk);
printf("Key Generation Result: %d\n", keygen_result);
if (keygen_result != 0) {
printf("Key generation failed.\n");
Py_Finalize();
return;
}
printf("Assinando mensagem...\n");
int sign_result = crypto_sign(signature, &siglen, message, MESSAGE_LEN, sk);
printf("Sign Result: %d\n", sign_result);
if (sign_result != 0) {
printf("Signing failed.\n");
Py_Finalize();
return;
}
printf("Verificando assinatura...\n");
int verify_result = crypto_sign_open(verified_message, &mlen, signature, siglen, pk);
printf("Verify Result: %d\n", verify_result);
if (verify_result != 0) {
printf("Verification failed.\n");
Py_Finalize();
return;
}
printf("Abrindo script Python ../test/quantum_attack_2.py...\n");
const char* script_path = "../test/quantum_attack_2.py";
FILE* fp = fopen(script_path, "r");
if (fp == NULL) {
printf("Failed to open %s\n", script_path);
Py_Finalize();
return;
}
printf("Executando script Python...\n");
PyRun_SimpleFile(fp, script_path);
fclose(fp);
if (PyErr_Occurred()) {
PyErr_Print();
}
Py_Finalize();
printf("Python finalizado.\n");
printf("Gerando chaves após ataque quântico...\n");
memset(pk, 0, PUBLIC_KEY_LEN);
memset(sk, 0, SECRET_KEY_LEN);
memset(signature, 0, SIGNATURE_LEN);
memset(verified_message, 0, MESSAGE_LEN);
```

```
        keygen_result = crypto_sign_keypair(pk, sk);
        printf("Key Generation Result After Quantum Attack: %d\n", keygen_result);
        printf("Assinando mensagem após ataque quântico...\n");
        sign_result = crypto_sign(signature, &siglen, message, MESSAGE_LEN, sk);
        printf("Sign Result After Quantum Attack: %d\n", sign_result);
        printf("Verificando assinatura após ataque quântico...\n");
        verify_result = crypto_sign_open(verified_message, &mlen, signature, siglen, pk);
        printf("Verify Result After Quantum Attack: %d\n", verify_result);
}
int main(void) {
        const struct CMUnitTest tests[] = {
        cmocka_unit_test(test_keygen),
        cmocka_unit_test(test_sign_verify),
        cmocka_unit_test(test_integration),
        cmocka_unit_test(test_edge_cases),
        cmocka_unit_test(test_randomized),
        cmocka_unit_test(test_known_answers),
        cmocka_unit_test(test_quantum_resistance),
        };
        return cmocka_run_group_tests(tests, NULL, NULL);
}
```

ANEXO II

```
try:
        from qiskit_aer import Aer
        from qiskit import transpile, QuantumCircuit

        # Simulador de circuito quântico
        simulator = Aer.get_backend('qasm_simulator')

        # Criação de um circuito quântico simples para exemplo
        circuit = QuantumCircuit(2, 2)
        circuit.h(0)
        circuit.cx(0, 1)
        circuit.measure([0, 1], [0, 1])

        # Transpilação do circuito
        transpiled_circuit = transpile(circuit, simulator)

        # Execução do circuito no simulador
        result = simulator.run(transpiled_circuit).result()

        # Resultado do experimento
        print(result.get_counts(circuit))

except ImportError as e:
        print(f"Erro ao importar o Qiskit: {e}")
        print("Certifique-se de que o Qiskit esteja instalado corretamente.")
        import sys
        sys.exit(1)
```