Segurança Computacional
Seminário

Eduardo Pereira de Sousa - 231018937
Luca Heringer Megiorin - 231003390
Nirva Neves de Macedo - 232009585
Universidade de Brasília

17 de Julho, 2025

1 Introdução

Desde a antiguidade, a necessidade de se validar fatos se faz presente na sociedade, sendo a assinatura a forma mais comum de se comprovar a veracidade de um documento e, com o surgimento e a expansão da computação, também tornou-se necessária a confirmação quanto à validade de documentos virtuais e outros arquivos. Para resolver esse problema, foi-se criada a assinatura digital, com o papel de checar a autenticidade e integridade do arquivo, confirmando a identidade do assinante e que o conteúdo não foi alterado, além de estabelecer o não-repúdio, ou seja, garantir que não é possível negar a autoria, dado que a assinatura é unicamente encarregada a um responsável.

Sendo assim, para o Seminário de Segurança Computacional, o grupo escolheu implementar um assinador e verificador de arquivos digitais, estudando também como sistemas reais funcionam. Neste documento, serão abordadas as escolhas de projeto (seção 2), detalhes sobre a interface (seção 2.2), questões de segurança do software (seção 2.3), o desempenho das funções (seção 3), e possibilidades de expansão do software, considerando softwares existentes (seção 4). O código desenvolvido em Python está disponível no GitHub (vide o apêndice A).

2 Escolhas de Projeto

O assinador e verificador digital foi implementado pelo grupo em Python, permitindo maior facilidade de criar uma aplicação local. O sistema é relativamente simples, permitindo a geração de chaves, que são guardadas localmente, a assinatura de um arquivo qualquer com uma dada chave privada e a verificação de um arquivo junto à sua assinatura, dada uma chave pública.

2.1 Algoritmos Usados

Por se tratar de uma assinatura digital, além do algoritmo para assinar, é necessário a criação de um hash. Seguindo as recomendações do NIST [1], foram escolhidos os algoritmos ECDSA e SHA3-512 para o programa, ambos presentes na biblioteca cryptography [2]. Quanto ao hash, foi escolhido o SHA3-512 por ser uma função robusta, já que gera um hash de 512 bits, ou seja, existe menos possibilidade de ocorrerem colisões.

Enquanto isso, no caso da escolha do algoritmo de assinatura, não se foi considerado o DSA por ser obsoleto, nem o EdDSA, dado que, apesar de ser o mais recente e eficiente, ainda não tinha o suporte de prehash na biblioteca escolhida, o que impossiblitaria realizar o hash em fluxo de um arquivo, ou seja, não permitiria que o sistema fizesse assinatura de arquivos muito grandes.

O RSA também foi evitado por já ter algumas vulnerabilidades conhecidas, como ataques ao padding ou em situações específicas co possuir algumas vulnerabilidades que permitem ataques a servidores TLS [3]. Apesar de o programa desenvolvido não se encaixar nos casos de possíveis vulnerabilidades do RSA, foi escolhido o ECDSA também por sua velocidade e maior confiança.

2.2 Interface e Modularidade

O programa foi implementado de forma modular, separando a interface das funções responsáveis pela segurança e processamento dos dados, o que permite uma fácil manutenção. Por simplicidade, não foi implementado um sistema de contas que permitisse a associação das chave a pessoas específicas, dado que o sistema teve um foco acadêmico para estudo do funcionamento de assinaturas digitais. Sendo assim, para a geração das chaves, é aberta uma janela solicitando uma senha (usada na serialização da chave privada - seção 2.3) e são geradas e serizalizadas as chaves públicas e privadas. Após serem geradas, as chaves são armazenadas, respectivamente, nas pastas *pub* e *priv* do projeto.

A partir disso, o usuário pode escolher um arquivo para assinar, precisando selecionar a sua própria chave privada e fornecer a sua senha escolhida no momento de gerar as chaves, o que garante que só quem sabe a senha possa usar tal chave (seção 2.3). Para a verificação, o usuário apenas necessita selecionar a chave pública do signatário, e, assim, o programa retornará se a assinatura está válida ou inválida, caso o documento original tenha sido alterado ou caso a chave pública não corresponda à chave privada que assinou o arquivo.

2.3 Segurança do Sistema

A segurança do processo de assinatura e verificação em si são apoiados pelas boas escolhas algorítmicas, baseadas nas recomendações da FIPS 186-5 [1] para assinatura digital, sendo que o algoritmo de curva elíptica, bem como a escolha do digest SHA3-512 fornecem bastante folga em questão das margens de segurança. Todavia, o sistema não opera com certificados digitais, ou seja, gera suas próprias chaves e as guarda em pastas

dentro do próprio aplicativo, o que pode representar uma vulnerabilidade, além de que o usuário pode escolher a chave privada que desejar.

Reconhecendo os riscos de segurança, realiza-se uma encriptação da chave privada em sua serialização para o arquivo PEM, também por meio da biblioteca cryptography [2], utilizando o método *BestAvailableEncryption*, que usa o melhor algoritmo simétrico disponibilizado pela OpenSSL. Com isso, mesmo que todos os usuários tenham acesso às chaves, não é possível usá-las sem saber da senha. Contudo, vale notar que não foram implementadas formas de garantir uma senha segura, como caracteres especiais e tamanho mínimo, então a segurança da chave privada é de responsabilidade total do usuário e da senha que ele escolher, o que não é bom em um contexto de ataques.

A necessidade, porém, de se trabalhar com uma senha dentro de um código Python, em que a presença do garbage collector pode trazer uma incerteza quanto ao descarte correto da senha após o seu uso. Para isso, não são usadas strings para manusear senhas, pois estas são imutáveis e reatribuir um valor à variável não sobrescreve a memória, apenas aloca espaço em outro lugar, o que pode ser uma vulnerabilidade no sistema. Ao invés de usar strings, todas as instâncias de senha no código são bytearrays, que são listas mutáveis de bytes, os quais podem ser sobrescritos com zeros em um loop, logo após o uso da senha naquela função ou parte do código. Dessa forma, a senha não fica no código por mais tempo do que o necessário para fazer as operações.

3 Desempenho

O desempenho do sistema de assinatura digital é um fator interessante, especialmente com a ideia de se poder assinar qualquer arquivo. Quanto a isso, é bom que o arquivo de assinatura seja separado do arquivo a ser verificado, tendo em vista que, como o arquivo de assinatura é pequeno, o processo de verificação fica ligeiramente mais rápido. Apesar de tudo, como dito anteriormente, isso é mais uma preocupação funcional, com o objetivo de possibilitar a assinatura de virtualmente qualquer coisa.

Relativo à esta questão, o projeto elaborado possui resultados satisfatórios para arquivos pequenos, considerando os dados apresentados e o tempo necessário para cada execução. Todavia, como é necessário fazer um *digest* (hash) do arquivo original tanto para a assinatura quanto para a verificação, em modo de fluxo de bytes, arquivos grandes (maiores que 1 GB) têm um tempo de processamento considerável, caso o programa não seja interrompido por erros (o que é possível ocorrer para arquivos muito grandes).

Deve-se ressaltar que, mesmo complexo, o módulo cryptography do Python é construído sobre a API em baixo nível da libcrypto e da libssl fornecidas pelo OpenSSL. Há inúmeros outros fatores que afetam o desempenho, e, provavelmente, o mais impactante deles é a leitura em fluxo dos arquivos. Foi observado que o hashing byte a byte do arquivo é significativamente mais lento que o hashing de bloco em bloco de 2 KiB (2048 bytes), principalmente para arquivos grandes. Esse efeito provavelmente se deve ao fato de que leituras repetidas da memória de armazenamento (SSD, HD) são lentas. Isso é de fato apenas uma distinção causada pela forma de acesso ao hardware, pois, mesmo assim, ambas as operações têm a mesma complexidade O(n).

Os testes foram feitos em um único computador, e é importante considerar que a utilização de uma biblioteca, apesar de segura, complexa, como contribuinte para o aumento do tempo de execução desses processos. Além disso, a dependência de uma linguagem interpretada como o Python para fins criptográficos também tem suas consequências operacionais no tempo, e ambos os fatores aqui mencionados podem ser vistos com o overhead gerado na primeira execução de da assinatura ou verificação de qualquer arquivo, causando um tempo extremamente maior do que o padrão obtido nas próximas 4 execuções.

Foram avaliados cinco tipos diferentes de arquivo(**.txt, .docx, .pdf, .png e .zip**). Apesar de serem tipos diferentes, o fator que mais causa alteração no tempo de execução é o tamanho do arquivo em si. Todos os arquivos foram assinados com a mesma chave privada e, consequentemente, verificados com a mesma chave pública.

3.1 Resultados da Assinatura e Verificação da Assinatura Pré-Otimização

	.txt	**.docx**	**.pdf**	**.png**	**.zip**
Tamanho do Arquivo	450 bytes	21.287 bytes	257.066 bytes	280.332 bytes	433.853 bytes
Execução 1	9.9607 ms	16.3004 ms	70.9041 ms	77.6118 ms	114.5072 ms
Execução 2	1.3779 ms	6.2209 ms	63.0500 ms	69.0575 ms	107.0349 ms
Execução 3	1.3670 ms	6.4054 ms	62.4625 ms	71.2963 ms	107.6181 ms
Execução 4	1.3874 ms	6.2748 ms	63.9943 ms	69.4736 ms	103.9301 ms
Execução 5	1.3722 ms	6.5056 ms	62.2599 ms	70.2394 ms	105.6117 ms
Média das 5 execuções	**3.09304 ms**	**8.34142 ms**	**64.53416 ms**	**71.53572 ms**	**107.7404 ms**
Média das execuções 2 – 5	**1.376125 ms**	**6.351675 ms**	**62.941675 ms**	**70.0167 ms**	**106.0487 ms**

Tabela 1: Tempo de 5 execuções de assinatura de cada formato de arquivo, com o tempo em negrito indicando a média das execuções

	.txt	**.docx**	**.pdf**	**.png**	**.zip**
Tamanho do Arquivo	450 bytes	21.287 bytes	257.066 bytes	280.332 bytes	433.853 bytes
Execução 1	13.0934 ms	11.8220 ms	76.4244 ms	82.6723 ms	118.7068 ms
Execução 2	0.6110 ms	5.5503 ms	62.6555 ms	67.7065 ms	105.1704 ms
Execução 3	0.5808 ms	5.6872 ms	61.6725 ms	67.7524 ms	106.3610 ms
Execução 4	0.5980 ms	6.3311 ms	61.3012 ms	67.9919 ms	105.9044 ms
Execução 5	0.6092 ms	5.6443 ms	60.8035 ms	76.4772 ms	104.3776 ms
Média das 5 execuções	**3.09848 ms**	**7.00698 ms**	**64.57142 ms**	**72.52006 ms**	**108.10404 ms**
Média das execuções 2 – 5	**0.59975 ms**	**5.803225 ms**	**61.608175 ms**	**69.982 ms**	**105.45335 ms**

Tabela 2: Tempo de 5 execuções de verificação da assinatura de cada arquivo, com o tempo em negrito indicando a média das execuções

3.2 Resultados da Assinatura e Verificação da Assinatura Pós-Otimização

Como dito anteriormente, após análise do código, foi possível otimizar drasticamente a eficiência do programa. Com a otimização do hashing de byte a byte (presente no branch main) para um hashing de 2048 em 2048 bytes (presente no branch de otimização), o processo foi acelerado, o que permitiu que o aplicativo fosse capaz de assinar arquivos de tamanho maior que antes sem interromper seu funcionamento. Em um teste com um arquivo .zip que ocupa 24GB de armazenamento, foi possível assinar o arquivo em 118 segundos e verificar sua assinatura em 123 segundos.

	.txt	**.docx**	**.pdf**	**.png**	**.zip**
Tamanho do Arquivo	450 bytes	21.287 bytes	257.066 bytes	280.332 bytes	433.853 bytes
Execução 1	9.1582 ms	9.3415 ms	10.0388 ms	10.3177 ms	14.0877 ms
Execução 2	1.3096 ms	1.4607 ms	2.4579 ms	2.6530 ms	3.2293 ms
Execução 3	1.2683 ms	1.4108 ms	2.4016 ms	2.5713 ms	3.2916 ms
Execução 4	1.2915 ms	1.4648 ms	2.5493 ms	2.5400 ms	3.3923 ms
Execução 5	1.4082 ms	1.4068 ms	2.4428 ms	2.5308 ms	3.2688 ms
Média das 5 execuções	**2.88716 ms**	**3.01692 ms**	**3.97808 ms**	**4.12256 ms**	**5.25394 ms**
Média das execuções 2 – 5	**1.3194 ms**	**1.435775 ms**	**2.4629 ms**	**2.573775 ms**	**3.0455 ms**

Tabela 3: Tempo de 5 execuções da assinatura de cada arquivo no projeto com hashes otimizados, com o tempo em negrito indicando a média das execuções

	.txt	**.docx**	**.pdf**	**.png**	**.zip**
Tamanho do Arquivo	450 bytes	21.287 bytes	257.066 bytes	280.332 bytes	433.853 bytes
Execução 1	7.6063 ms	6.8220 ms	9.3522 ms	8.6250 ms	16.9550 ms
Execução 2	0.5031 ms	0.6152 ms	1.6688 ms	1.7662 ms	2.4695 ms
Execução 3	0.4845 ms	0.5968 ms	1.6591 ms	1.7524 ms	2.4366 ms
Execução 4	0.5216 ms	0.6053 ms	1.6516 ms	1.9673 ms	2.4814 ms
Execução 5	0.5316 ms	0.5908 ms	1.6636 ms	1.7463 ms	2.4339 ms
Média das 5 execuções	**1.92942 ms**	**1.84602 ms**	**3.19906 ms**	**3.17144 ms**	**5.35528 ms**
Média das execuções 2 – 5	**0.5102 ms**	**0.602025 ms**	**1.660775 ms**	**1.80805 ms**	**2.45535 ms**

Tabela 4: Tempo de 5 execuções de verificação da assinatura de cada arquivo no projeto com hashes otimizados, com o tempo em negrito indicando a média das execuções

4 Possibilidades de Expansão

De fato, um simples verificador de assinatura local é um software vulnerável em um contexto mais amplo. Apenas gerar suas chaves, sem ter acesso a uma Autoridade Certificadora, ou mesmo a simples keyservers, comumente utilizados para verificar arquivos de

software (como imagens de sistemas operacionais), para autenticá-las enfraquece a utilidade do programa desenvolvido sendo aplicado em situações reais. Almejando expandir o software além do contexto acadêmico, e com um olhar para o mercado, seria possível se inspirar no software GnuPG, um pacote que fornece facilidades para criptografia e assinatura para comunicação de dados, e poder-se-ia implementar um gerenciamento automático de chaves, como, por exemplo, a possibilidade de recolher chaves públicas de um servidor específico, utilizando APIs seguras para este fim.

Além disso, como arquivos maiores levam de alguns segundos até minutos para a finalização da assinatura/verificação, seria interessante incluir novos elementos na interface como, por exemplo, uma barra de progresso, que seria uma adição simples, mas suficientemente confiável, já que as operações de assinar e verificar são, em essência, apenas uma leitura e hashing O(n) do arquivo. Uma interface mais amigável também forneceria possibilidades mais amplas de aplicação. De fato, essas considerações vão além do escopo do projeto em si, mas aplicariam os conceitos da disciplina de Segurança Computacional de forma extremamente abrangente.

Por fim, por se tratar de um projeto acadêmico, para o qual o tempo disponível para sua realização foi curto, a escolha da linguagem de programação Python foi necessária. No entanto, fora desse contexto, uma boa escolha de linguagem compilada, mais robusta, seria, por exemplo, a linguagem Rust, que fornece medidas contra vulnerabilidades relacionadas a vazamentos de memória sem a necessidade de um coletor de lixo, abrindo caminhos para o desenvolvimento de um software criptográfico rápido e seguro, em contraste à linguagem C, que dificulta a análise de segurança.

5 Conclusão

Com este seminário, foi possível entender melhor sobre as assinaturas digitais, o seu funcionamento e a dificuldade do gerenciamento de chaves. A implementação de um sistema de assinatura e verificação permitiu que fossem aplicados os conceitos teóricos vistos em sala de aula em um contexto mais tangível, em que era necessário levar em conta as vulnerabilidades de um software quanto à sua segurança. O código em Python usado para o seminário e os slides podem ser acessados no GitHub, com o link no apêndice A. Vale notar que há um branch com o código otimizado, como mencionado anteriormente.

Referências

[1] NIST Digital Signature Standard (DSS), FIPS 186-5. Disponível em: <https://csrc.nist.gov/pubs/fips/186-5/final>. Acesso em: 16 de julho de 2025.

[2] Biblioteca Cryptography para o Python. Disponível em: <https://cryptography.io/>.

[3] The ROBOT Attack. Disponível em: <https://robotattack.org>.

A Código do Seminário e Material

O código em Python 3.15.5 e o material usado para o seminário (slides) estão presentes no link do GitHub abaixo:

- Link para repositório: https://github.com/Luke0133/SeminarioSC/tree/main.