# Instructions for Authors of SBC Conferences Papers and Abstracts

**Giovani Ferreira**[1], **Rafael Marconi**[1]

[1]CEUB - Centro Universitário de Brasília
Caixa Postal 4488 – 70.904-970 – Brasília – DF – Brazil

***Abstract.*** *adhaskjdhaksjd [Tanenbaum and Van Steen 2002]*

## 1. Introdução

Na era em que computadores e dispositivos eletrônicos tem assumido um papel crucial, tornou-se muito importante manter a segurança e integridade de informações transmitidas através desses dispositivos. Para alcançar esse objetivo, é amplamente adotado o uso de funções hash, obtendo a integridade da seguinte maneira: Em algum ponto do tempo é computado o valor hash correspondente a uma entrada particular. Em outro ponto subsequente, para verificar que o dado de entrada não foi alterado, o valor hash é computado novamente usando a entrada disponível e comparado por igualdade com o valor original. Aplicações específicas de função hash incluem a proteção contra vírus e distribuição de software [Menezes et al. 1996].

Esse cenário apresenta uma característica importante. Se um adversário consegue alterar a entrada e mesmo assim obter um valor do hash igual ao valor da entrada original, o destinatário entenderá que os dados recebidos são íntegros. Tal ataque foi demonstrado em diferentes trabalhos [Lenstra et al. 2005], [Stevens et al. 2007] e [Lenstra and De Weger 2005], que provam ser possível alterar um certificado X.509, inclusive a chave pública, e obter o mesmo valor hash.

Essa busca por colisão, ou seja, dois valores distintos que possuam o mesmo valor hash, é uma ferramenta importante em criptoanálise. Uma gama grande de problemas criptoanalíticos tais como computar logaritmos discretos, achar colisões de função hash e o ataque por encontro a médio caminho, meet-in-the-middle, podem ser reduzidos ao problema de achar duas entradas distintas, $a$ e $b$, para uma função $f$ de forma que $f(a) = f(b)$ [Van Oorschot and Wiener 1999]. Portanto, para garantir a segurança, é necessário que funções hash possuam a propriedade de ser computacionalmente inviável achar duas entradas diferentes que produzam o mesmo valor hash.

(Parágrafo para falar que os computadores são potentes e introduzir os clusters raspberry) Por outro lado, os recursos tecnológicos em constante evolução possibilitam um poder computacional maior. Com processadores capazes de realizar milhões de operações por segundo

Utilizando desse poder computacional de baixo custo e de princípios básicos de probabilidade (paradoxo do aniversário), é possível desenvolver um ataque de força bruta que consiga encontrar uma colisão em funções hash. Tal ataque é denominado ataque do aniversário (Birthday Attack) e consiste, em sua forma mais simples, de escolher um quantidade $n$ de mensagens aleatórias, computar seus valores hash $h_1, h_2, ..., h_n$ e testar por colisão entre todos, ou seja, $h_i = h_j$ para $i, j \in 1, 2, ..., n$.

Apesar de técnicas para prevenir ataques baseados no paradoxo do aniversário já terem sido apresentadas [Aiello and Venkatesan 1996], é interessante discutir o ataque do aniversário já que é possível observar como simples ideias matemáticas em conjunto com um cluster de microprocessadores de baixo custo podem ser aplicadas e usadas como forma de ataque cibernético.

Dessa forma, este artigo consiste de cinco seções. A segunda tratará da explicação e definição de conceitos relacionados, como função hash e colisão de valores hash. A terceira consistirá em detalhar e abordar a implementação do cluster de raspberry e dos testes serial, paralelo e distribuído. A quarta seção irá expor os resultados obtidos a partir dos testes e a comparação de performance entre eles. E, por fim, a quinta seção dissertará sobre as conclusões obtidas através dos testes.

## 2. Conceitos Relacionados

### 2.1. Função Hash

Uma função hash é uma função computacionalmente eficiente $f$ que mapeia uma sequência binária de tamanho arbitrário $\{0,1\}^n$ para uma sequência binária de tamanho fixo $\{0,1\}^x$, chamada de valor hash, hash ou digest [Menezes et al. 1996]. Ou seja, $f : \{0,1\}^n \rightarrow \{0,1\}^x$ para $n \gg x$ e $x$ igual a um valor fixo como $128, 160, 256$ e $512$.

Tipicamente, funções hash são construídas a partir de uma função $h : B \times R \rightarrow R$, que recebe um bloco de tamanho fixo de uma mensagem junto com um valor hash intermediário e produz um novo valor hash intermediário. Uma mensagem recebida $m \in \{0,1\}^n$ é tipicamente completada para ter um comprimento que seja múltiplo do tamanho do bloco. Em sequência é quebrada em blocos $m_1, m_2, ..., m_k \in B$ e começando com alguma constante $r_0 \in \mathbb{R}$, a sequência $r_i = h(m_i, r_{i-1})$ é computada para $i = 1, 2, ..., k$ e $r_k$ é o resultado do hash para a mensagem $m$ [Van Oorschot and Wiener 1999].

### 2.2. Colisão Hash

Funções hash são projetadas para receber uma mensagem de tamanho arbitrário e mapear ela para uma saída de tamanho definido. Se o conjunto de entrada $\mathbb{M}$ é muito maior que o conjunto de saída, $\mathbb{M} \gg \mathbb{R}$, então existem valores $m \in \mathbb{M}$ que irão ser mapeados para um mesmo valor em $\mathbb{R}$. Uma colisão acontece quando encontra-se dois valores $m_1$ e $m_2$ sendo que $m_1 \neq m_2$ mas $h(m_1) = h(m_2)$.

Uma função hash $h$ é chamada de livre de colisão ($collision\ free$) ou resistente a colisão, se ela mapeia mensagens de qualquer tamanho para strings de tamanho definido, mas é computacionalmente inviável achar $x, y \mid h(x) = h(y)$ [Damgård 1989] [Menezes et al. 1996].

### 2.3. Paradoxo do Aniversário

O paradoxo do aniversário é o princípio contraintuitivo que para grupos de somente $23$ pessoas, existe a chance de aproximadamente $50\%$ de encontrar duas pessoas com o mesmo aniversário (Assumindo que todos os aniversários são igualmente prováveis e desconsiderado anos bissexto). Comparado a probabilidade de encontrar alguém nesse grupo com o mesmo aniversário que o seu, aonde se têm $23$ chances independentes e portanto uma probabilidade de sucesso de $\frac{23}{365} \approx 0.06$, esse princípio é baseado no fato

de que existe $\frac{23*22}{2} = 253$ pares distintos de pessoas. Isso leva a uma probabilidade de sucesso por volta de $0.5$ (note que isso não é igual a $\frac{253}{365} \approx 0.7$ já que esses pares não são independentemente distribuídos) [Stevens et al. 2012].

Uma conclusão importante obtida disso é que se uma função hash gera como saída $n$ bits, sempre haverá um ataque que roda em tempo $2^{\frac{n}{2}}$. Por exemplo, se a saída é de $128$ bits, então uma colisão pode ser encontrada em tempo $\approx 2^{64}$, o que não é considerado suficientemente seguro. Outra observação interessante é que aniversários na verdade não são uniformemente distribuídos, mas o paradoxo do aniversário foi montado com a suposição de que eles são uniformemente distribuídos.

### 2.4. Ataque do Aniversário

O ataque do aniversário é a aplicação do paradoxo do aniversário em um cenário real, buscando utilizar desse princípio como uma forma ofensiva de obter, burlar ou alterar informações restritas. No contexto deste artigo, o ataque do aniversário consistirá na aplicação do princípio para encontrar duas mensagens diferentes que possuam o mesmo valor hash. Sabendo que o conjunto das $tags$ é igual a $\{0,1\}^n$ e dada a função hash $h : \mathbb{M} \rightarrow \{0,1\}^n$, sendo que $\mid tags \mid \approx 2^n$ bits e que $|\mathbb{M}| \gg 2^n$, o algoritmo genérico para o ataque do aniversario é composto pelas seguintes etapas:

1. Escolhe-se $2^{\frac{n}{2}}$ mensagens aleatórias em $\mathbb{M}$, de forma que $m_1, m_2, ..., m_{2^{\frac{n}{2}}} \in \mathbb{M}$;
2. Para $i = 1, 2, ..., 2^{\frac{n}{2}}$ computa-se $t_i = h(m_i)$, aonde $t_i$ é o valor hash no conjunto das $tags$, ou seja, $t_i \in tags$;
3. Busca-se por qualquer colisão, isto é, $t_i = t_j$ para $i, j \in 1, 2, ..., 2^{\frac{n}{2}}$. Caso não seja encontrada, volta-se a etapa $1$ e repete-se com uma amostra diferente de mensagens.

## 3. Implementação do Ataque do Aniversário

### 3.1. Estruturação do Cluster Raspberry

### 3.2. Implementação do Ataque

A implementação do ataque foi efetuada de seis maneiras diferentes. Consistindo de duas implementações seriais, duas paralelas e duas distribuídas. As implementações obrigatoriamente tem que realizar três tarefas diferentes:

- Coletar mensagem aleatória $m \in \mathbb{M}$;
- Computar o valor hash $h_m$ da mensagem;
- Comparar o valor hash com os outros valores computados.

Elas diferem entre si pela forma como executam essas tarefas. Sendo ou modular, aonde as tarefas são executadas separadamente, ou contínua, aonde um pedaço de cada tarefa é executado repetidamente até todas as tarefas serem concluídas.

#### 3.2.1. Implementação Serial Contínua

A implementação serial foi planejada para executar em um único core de um processador. Sendo feita sequencialmente e sem distribuição do pro-

cessamento. O algoritmo serial contínuo funciona da seguinte maneira:

```
n = número de mensagens aleatórias;
while Primeiro iterador for menor que n do
    Coleta uma mensagem aleatória;
    Computa o valor hash correspondente a mensagem;
    while Segundo iterador for menor que o número de hashs já computados
     do
        Compara o hash atual com os outros já computados;
    end
end
```

### 3.2.2. Implementação Serial Modular

A implementação serial modular possui as mesmas características da serial contínua, com exceção da sequência em que as tarefas ocorrem. Elas são executadas separadamente:

```
n = número de mensagens aleatórias;
while Iterador < n do
    Coleta uma mensagem aleatória;
    Computa o valor hash correspondente a mensagem;
end
while Iterador < número de hashs já computados do
    Compara o hash atual com os outros;
    if Hash atual = Outro hash then
        Colisão encontrada;
        Aborta a execução;
    end
end
```

### 3.2.3. Implementação Paralela Contínua

### 3.2.4. Implementação Paralela Modular

### 3.2.5. Implementação Distribuída Contínua

### 3.2.6. Implementação Distribuída Modular

## 4. Experimentos e Resultados

## 5. Conclusões e Trabalhos Futuros

## Referências


Aiello, W. and Venkatesan, R. (1996). Foiling birthday attacks in length-doubling transformations. In *Advances in Cryptology—EUROCRYPT'96*, pages 307–320. Springer.

Damgård, I. B. (1989). A design principle for hash functions. In *Advances in Cryptology—CRYPTO'89 Proceedings*, pages 416–427. Springer.

Lenstra, A. and De Weger, B. (2005). On the possibility of constructing meaningful hash collisions for public keys. In *Australasian Conference on Information Security and Privacy*, pages 267–279. Springer.

Lenstra, A. K., Wang, X., and de Weger, B. (2005). Colliding x. 509 certificates. Technical report.

Menezes, A. J., Van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of applied cryptography*. CRC press.

Stevens, M., Lenstra, A., and De Weger, B. (2007). Chosen-prefix collisions for md5 and colliding x. 509 certificates for different identities. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–22. Springer.

Stevens, M. M. J. et al. (2012). *Attacks on hash functions and applications*. Mathematical Institute, Faculty of Science, Leiden University.

Tanenbaum, A. S. and Van Steen, M. (2002). *Distributed systems: principles and paradigms*, volume 2. Prentice hall Englewood Cliffs.

Van Oorschot, P. C. and Wiener, M. J. (1999). Parallel collision search with cryptanalytic applications. *Journal of cryptology*, 12(1):1–28.