Invariantes Modulares, Funções Não-Invertíveis e Criptografia de Curva Elíptica Aplicada a Sistemas de Pagamento

Quando se fala em segurança de cartões de crédito, gateways de pagamento e autenticação de senhas, a primeira impressão de muitos é que há uma espécie de “mágica” acontecendo: você digita seu número, sua senha, o cartão passa na máquina e pronto, você é autorizado ou não. A verdade é que não existe magia alguma; existe matemática aplicada de forma engenhosa. O que diferencia sistemas confiáveis de sistemas vulneráveis é o entendimento profundo de invariantes matemáticos, congruências, funções não-invertíveis e operações criptográficas. Sistemas modernos **não verificam segredos**, eles verificam **propriedades que só os elementos corretos podem satisfazer**. E é isso que garante que, mesmo sem conhecer sua senha ou sua chave privada, o sistema saiba se sua entrada está correta.

Para começar a entender essa dinâmica, é essencial voltar no tempo até 1954, quando Hans Peter Luhn, engenheiro da IBM, propôs um algoritmo para validar números de cartões de crédito. O algoritmo de Luhn é aparentemente simples, mas revela uma profundidade matemática elegante quando analisado do ponto de vista de teoria dos números. A ideia central é transformar uma sequência de dígitos *d1, d2 ... dn* em um somatório ponderado *S* tal que *S ≡ 0 mod (10)*. Formalmente seja:

$$S = \sum_{i=1}^{n} f(di), S = 0(mod10)$$

Onde a função *f(di)* aplica um peso alternado (normalmente dobrando dígitos alternados, subtraindo 9 se o resultado for maior que 9). O número é aceito se *S* pertencer à classe de congruência 0 módulo 10. Aqui já se vê um padrão fundamental: **não se verifica o número em si, mas se ele satisfaz uma condição estrutural invariável**. Essa ideia de “verificar uma propriedade e não o objeto” é a base de toda a criptografia aplicada em cartões.

Do ponto de vista da computação, implementar Luhn é trivial, mas revela a beleza da abstração: cada dígito é processado individualmente, com multiplicações e somas, e depois testamos a congruência. Em C, o algoritmo fica assim:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int luhn(const char *num) {
    int sum = 0, alt = 0;
    for (int i = strlen(num) - 1; i >= 0; i--) {
        if (!isdigit(num[i])) return 0;
        int n = num[i] - '0';
        if (alt) {
```

```
            n *= 2;
            if (n > 9) n -= 9;
        }
        sum += n;
        alt = !alt;
    }
    return sum % 10 == 0;
}

int main() {
    const char *cartao = "4539148803436467";

    if (luhn(cartao)) {
        printf("Valid number\n");
    } else {
        printf("Invalid number\n");
    }

    return 0;
}
```

Rant

> *Obs:. eu deixei o numero com 16 dígitos simulando um número de cartão apenas para exemplificar sem exigir a necessidade de usar por conta própria um numero com 16 dígitos válido. esse número é matematicamente feito para sempre ser "Valid Number" apenas com o intuito de exemplificar tal feito.*

> *Obs II:. a função após o **Algorítmo de Luhn** é apenas para verificar o que foi feito na função **luhn***

Em R, podemos trabalhar de forma vetorial e aplicar simulações em larga escala, ideal para explorar comportamentos estatísticos de dígitos e validar heurísticas de entrada:

```
luhn_vector <- function(numero) {
  d <- as.integer(strsplit(numero, "")[[1]])
  d <- rev(d)
  weights <- rep(c(1,2), length.out = length(d))
  d_weighted <- d * weights
  d_weighted <- ifelse(d_weighted > 9, d_weighted - 9, d_weighted)
  sum(d_weighted) %% 10 == 0
}
```

O interessante é que o Luhn não garante que o cartão exista; ele apenas filtra entradas inválidas. É uma **verificação estrutural, não ontológica**. Essa mesma lógica se estende para senhas, mas com maior complexidade. Um sistema de autenticação nunca armazena

sua senha x. Ele armazena $H(x)$, onde H é uma função hash criptográfica resistente a colisões. Para verificar se a entrada x' está correta, o sistema testa:

$$H(x') = H(x)$$

O sistema nunca precisa conhecer x, apenas se a função hash aplicada à entrada corresponde ao invariante armazenado. Novamente, vemos a ideia central: **verificar uma propriedade invariável, não o segredo em si**.

Com a introdução dos cartões com chip, essa lógica evoluiu para curvas elípticas e funções de difícil inversão. Protocolos como ECDSA (Elliptic Curve Digital Signature Algorithm) e ECDH (Elliptic Curve Diffie-Hellman) aplicam operações matemáticas sobre grupos abelianos finitos definidos por equações da forma $y^2 = x^3 + ax + b$ sobre corpos finitos. A assinatura digital de um cartão envolve gerar (r , s) tal que:

$$r = (kG)_x mod n, s = k^{-1}(h + xr) mod n$$

O verificador utiliza apenas a chave pública $Q = kG$ e a assinatura (r , s) para validar se a equação é satisfeita. O segredo k nunca é revelado; apenas a **propriedade matemática que só k satisfaz** é verificada. Em ECDH, o segredo compartilhado $S = KaKbG$ é calculado usando apenas valores públicos, garantindo que apenas as chaves corretas produzam o mesmo resultado. Isso é a materialização prática da frase: $f(x) = c$ **sem conhecer** x.

Em R, podemos simular a verificação de invariantes para estudar comportamento de chaves, assinaturas e hashes:

```r
library(digest)
hash_check <- function(senha, hash_armazenado) {
  digest(senha, algo="sha256") == hash_armazenado
}

simulate_attack <- function(G_order, attempts=1e6, known_invariant) {
  successes <- 0
  for(i in 1:attempts) {
    x <- sample(1:G_order, 1)
    y <- sample(1:G_order, 1)
    if((x*y) %% G_order == known_invariant) successes <- successes + 1
  }
  successes / attempts
}
```

A partir daqui, a matemática se intensifica: a escolha de curvas elípticas e parâmetros de segurança está ligada à distribuição de primos e à dificuldade do logaritmo discreto, conceitos analisados por Ramanujan em suas heurísticas. Esses princípios garantem que, mesmo com acesso à função e aos resultados públicos, inverter o processo é computacionalmente inviável.

Do ponto de vista de engenharia de software, gateways de pagamento aplicam filtros em camadas: Luhn verifica consistência estrutural, hashes e MACs verificam integridade e

autenticação, ECDSA/ECDH garantem segurança criptográfica e autenticidade do cartão. Em cada nível, **os segredos nunca são expostos**; apenas invariantes são checados.

A conclusão é que toda a segurança do fluxo de pagamento moderno repousa sobre a matemática: ***verificar invariantes é suficiente para autenticar entradas, senhas e cartões***, sem revelar os segredos subjacentes.

Até aqui estabelecemos a premissa central: *sistemas modernos não precisam conhecer segredos, apenas verificam **invariantes matemáticos***. Começamos pelo Luhn e abordamos superficialmente ECDSA/ECDH porém ficou buracos em branco. Agora, é hora de preencher as lacunas e tentar entender **como esses mecanismos se articulam em conjunto**, desde o ato de inserir um cartão na maquininha até a validação final no servidor bancário;

> ***disclaimer**:. não espere que esse curto artigo seja uma forma de solucionar e explicar tudo que acontece, isso seria parcialmente improvável de ocorrer.**

Quando você insere um cartão, seja físico ou virtual, o dispositivo inicia uma sequência de operações que podem ser interpretadas como **uma cadeia de verificação de invariantes**. Primeiro, o número do cartão passa pelo Luhn, garantindo integridade sintática. Em seguida, o chip realiza operações criptográficas locais. Para ECDSA, o cartão gera uma assinatura ***(r , s)*** sobre uma mensagem *m* (que pode incluir o valor da transação, timestamp e identificação do terminal). Formalmente:

$$r = (kG)_x mod n, s = k^{-1}(h(m) + xr) mod$$

O terminal recebe ***(r , s)*** e, com a chave pública ***Q = kG*** verifica se:

$$s^{-1}h(m)G + s^{-1}rQ = rG$$

Se a equação fecha, a assinatura é válida e o cartão é autenticado. Note que **nenhuma chave privada foi exposta**, apenas a relação invariável foi satisfeita. Essa é a razão pela qual a comunicação segura em gateways não precisa “conhecer sua senha”, tudo é transformado em invariantes.

Além disso, em protocolos de troca de chave como ECDH, duas partes (terminal e banco) derivam um segredo compartilhado $S = K_a K_b G$ usando apenas chaves públicas. A segurança desse processo depende da dificuldade do **logaritmo discreto em curvas elípticas**, ou seja, não existe algoritmo eficiente para recuperar *Ka* ou *Kb* apenas conhecendo *S* e *G*.

Do ponto de vista computacional, é possível simular essas operações em R para entender probabilidades de colisão ou sucesso de ataque de força bruta. Por exemplo, podemos

calcular a taxa de sucesso de tentativa de adivinhação de chaves simplificadas:

```
simulate_ecc <- function(G_order, attempts=1e6, secret_invariant) {
  successes <- 0
  for(i in 1:attempts) {
    k1 <- sample(1:G_order, 1)
    k2 <- sample(1:G_order, 1)
    derived <- (k1 * k2) %% G_order
    if(derived == secret_invariant) successes <- successes + 1
  }
  successes / attempts
}
```

Essa simulação não é um ataque real, mas permite **experimentar como a complexidade aumenta exponencialmente** com o tamanho do grupo. É aqui que heurísticas inspiradas em Ramanujan entram: *ao escolher curvas e parâmetros primos adequados, garantimos que **G** tenha ordem grande e que os números usados como chaves sejam distribuídos de forma quase uniforme, dificultando qualquer ataque baseado em tentativa e erro*.

Voltando à perspectiva de engenharia de software, cada camada do gateway ou terminal de pagamento aplica verificações diferentes:

- **Integridade sintática**: Luhn garante que o cartão foi digitado corretamente
- **Autenticação local**: MACs e ECDSA verificam que o cartão possui a chave privada correta sem expô-la.
- **Autenticação remota**: Hashes e assinaturas digitais enviadas ao banco garantem que a mensagem (transação) não foi alterada no caminho.
- **Validação final**: O banco valida invariantes globais, como saldo, limite de crédito e consistência de assinatura.

Cada uma dessas etapas utiliza **funções não-invertíveis**, congruências e propriedades de grupos finitos para reduzir a chance de erro humano e ataque computacional. Em termos de programação, podemos imaginar um fluxo simplificado em R integrando essas camadas:

```
validate_transaction <- function(card_number, password, transaction_value,
hash_db, ecc_params) {

  # aqui fazemos a checagem / Here we do the check.
  if(!luhn_vector(card_number)) return(FALSE)

  # verificação de senha / password verification occurs
  if(!hash_check(password, hash_db)) return(FALSE)

  # simulação ECDSA / ECDS simulation
  signature_valid <- simulate_ecc(ecc_params$G_order, ecc_params$attempts,
ecc_params$secret_invariant)
  if(signature_valid < ecc_params$threshold) return(FALSE)
```

```
  return(TRUE)
}
```

A abordagem modular feita mostra como a matemática se traduz diretamente em **engenharia prática**, cada etapa é uma *camada de invariantes* que garante a validade do sistema sem precisar conhecer o segredo do usuário.

Outro ponto que é de suma importancia é a análise de **robustez matemática**. Quanto maior a ordem do grupo elíptico, maior a complexidade de quebrar o sistema por força bruta. Curvas seguras com heurísticas de Ramanujan garantem que os elementos primos escolhidos como parâmetros tenham propriedades ideais de distribuição, evitando fraquezas estruturais. Em termos simplificados, estamos escolhendo conjuntos de chaves que maximizam a **entropia e a uniformidade**, assegurando que qualquer tentativa de adivinhação seja estatisticamente irrelevante.

Ao unirmos todos esses elementos (Luhn, hashes, MACs, ECDSA/ECDH), heurísticas de primos, entendemos que **o segredo nunca precisa ser exposto**. Sistemas modernos simplesmente verificam se o objeto que você forneceu satisfaz **todas as propriedades invariantes esperadas**. É por isso que, quando você insere um cartão e digita sua senha, o terminal sabe se está correto, mesmo que **nem o gateway nem o banco conheçam seu segredo em forma explícita**.

Desde a camada de entrada (Luhn), passando por autenticação local (MAC/assinatura), até validação remota (banco e gateway), a segurança é uma construção baseada em **matemática aplicada**, e não em segredo compartilhado. Essa é a beleza do design de sistemas de pagamento moderno.

ECDSA e ECDH

Comecemos desfazendo um erro comum e eu gosto de refoçar: ***ECDSA e ECDH* não são "algoritmos de criptografia" no sentido clássico**. Eles não "escondem" mensagens. Eles constroem **relações verificáveis** em um espaço algébrico onde a inversão é computacionalmente inviável.

Considere uma curva elíptica definida sobre um corpo finito:

$$E : y2 \equiv x3 + ax + b(modp)$$

onde *p* é primo grande. O conjunto de pontos ***E(Fp)***, junto com o ponto no infinito, forma um grupo abeliano finito.

Existe um ponto gerador ***G*** de ordem ***n***. A chave privada é um inteiro:

$$k \in 1, \ldots, n-1$$

e a chave pública é:

$$Q = kG$$

Aqui está o primeiro invariante fundamental:

> Conhecer *G* e *Q* **não** permite recuperar *k* em tempo viável.

Isso é **Elliptic Curve Discrete Logarithm Problem (ECDLP)**.

ECDH: acordo sem troca de segredo?

Duas partes escolhem chaves privadas: K_a e K_b

$$Q_A = k_A G, QB = k_B G$$

o segredo compartilhado é:

$$S = k_A Q_B = k_B Q_A = k_A k_B G$$

O sistema nunca verifica *Ka* e *Kb*. Ele verifica ***a igualdade estrutural do ponto derivado***.

ECDSA: provar posse sem revelar?

Na assinatura ECDSA, o cartão (ou chip) escolhe um nonce aleatório *r*

$$R = rG$$

Define-se:

$$s \equiv r^{-1}(h(m) + k \cdot R_x)(mod n)$$

A assinatura é o par (R_x, s)

A verificação não tenta recuperar *k*. Ela testa se a seguinte identidade se sustenta:

$$s^{-1}h(m)G + s^{-1}R_x Q = R$$

Simulações em R: por que força bruta não escala

Vamos reduzir artificialmente o problema para algo simulável e observar o crescimento da complexidade, no caso, na linguagem R

```r
simulate_dlp <- function(group_order, attempts = 1e6) {
  secret <- sample(1:group_order, 1)
  success <- FALSE
```

```
  for (i in 1:attempts) {
    guess <- sample(1:group_order, 1)
    if (guess == secret) {
      success <- TRUE
      break
    }
  }
  success
}
```

Se n = 10^6, força bruta é trivial.
Se n ≈ 2^{256}, a simulação se torna conceitualmente absurda.

A segurança **não vem do algoritmo**, mas da **ordem do grupo e da distribuição dos elementos**

Heurísticas inspiradas em Ramanujan e escolha de parâmetros

Vamos lá, Ramanujan não trabalhava com criptografia, mas com **distribuição profunda de números e padrões não óbvios**. O espírito das heurísticas modernas inspiradas nele é simples:

> Escolher estruturas matemáticas onde **regularidades locais não revelam estrutura global**.

Seria:

- Primos grandes com propriedades específicas
- Curvas elípticas sem subgrupos pequenos
- Ordens de grupo com fatoração resistente

Curvas mal escolhidas introduzem **atalhos algébricos**. Curvas bem escolhidas fazem com que qualquer tentativa de exploração pareça ruído estatístico.

Referências

Luhn, H. P. (1954). *Computer for Verifying Numbers*. IBM Technical Disclosure Bulletin.
Knuth, D. E. (1998). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley.
ISO/IEC 7812. *Identification cards — Identification of issuers*.
Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press.
Katz, J., & Lindell, Y. (2014). *Introduction to Modern Cryptography*. CRC Press.
Koblitz, N. (1987). *Elliptic Curve Cryptosystems*. Mathematics of Computation.
Miller, V. S. (1986). *Use of Elliptic Curves in Cryptography*. Advances in Cryptology —

CRYPTO ’85.
NIST FIPS 186-4. *Digital Signature Standard (DSS)*.
SEC 1. *Elliptic Curve Cryptography*. Standards for Efficient Cryptography Group (SECG).
EMVCo. *EMV Integrated Circuit Card Specifications for Payment Systems*.
Anderson, R. (2020). *Security Engineering*. Wiley.
Apple Inc. *Apple Pay Security and Privacy Overview*.
Google. *Google Pay API Security Whitepaper*.
Arora, S., & Barak, B. (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press.
Boneh, D., & Shoup, V. (2020). *A Graduate Course in Applied Cryptography*.
Ramanujan, S. (1927). *Collected Papers of Srinivasa Ramanujan*. AMS.
Granville, A. (2008). *Harald Cramér and the distribution of prime numbers*.
Soundararajan, K. (2009). *The distribution of prime numbers*.
Schneier, B. (1996). *Applied Cryptography*. Wiley.

Ferramentas

LaGrida LaTeX Editor. Online LaTeX equation editor. Avaliable at: https://latexeditor.lagrida.com/. Access at: 2026.
Linguagem C, conforme o padrão ISO/IEC 9899.
R Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing. Available at: https://www.R-project.org/.