UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Marina da Silva Coelho

QUEBRA-CABEÇAS CRIPTOGRÁFICOS

Florianópolis

2017

Marina da Silva Coelho

QUEBRA-CABEÇAS CRIPTOGRÁFICOS

Trabalho de Conclusão de Curso submetido ao curso de Graduação em Sistemas da Informação para a obtenção do Grau de Bacharel em Sistemas da Informação.
Orientador: Prof. Dra. Lucia Rosana Moura
Coorientador: Prof. Dr. Ricardo Felipe Custódio

Florianópolis

2017

Marina da Silva Coelho

QUEBRA-CABEÇAS CRIPTOGRÁFICOS

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Sistemas da Informação", e aprovado em sua forma final pelo curso de Graduação em Sistemas da Informação.

Florianópolis, 01 de novembro 2017.

Prof. Dr. Cristian Koliver
Coordenador do Curso

Prof. Dra. Lucia Rosana Moura
Orientador

Prof. Dr. Ricardo Felipe Custódio
Coorientador

Banca Examinadora:

Prof. Dra. Jerusa Marchi

Alex Sandro da Silva Pereira

Dedico este trabalho à minha eterna e amada Vovó Maria, por todo o apoio e incentivo que me deu em vida, e por continuar iluminando meu caminho lá de cima

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus por caminhar ao meu lado em todos os momentos e me proporcionar uma família tão maravilhosa. Aos meus pais e meu irmão, que aguentaram as crises de choro, a ausência nos fins de semana, o medo de dar tudo errado, e continuaram firmes e fortes, dando o melhor deles para que eu me sentisse segura.

Agradeço também aos meus queridos padrinhos e minha prima Sofia, por serem pais e irmã de coração, investindo longas horas de conselhos e conversas para me ajudar, fazendo com que a distância física entre nós parecesse inexistente. Aos meus amados avós, Vô Neri, Vô Zezinho e Vó Nina, por sempre demonstrarem tanto carinho e preocupação perguntando: "E os estudos, como vão?". Impossível esquecer da minha tia Goreti! Tia, obrigada por me acolher nestas últimas fases e por me acompanhar nos baldes de pipocas que me aguentaram acordada para finalizar este trabalho.

A todos os meus familiares e aos meus amigos de verdade, especialmente Henrique Prandi, por me ensinar a prestar atenção no agora, deixando de lado os medos e as inseguranças, e por ter feito estes anos de universidade ainda mais especiais, e Daniel Yoshizawa por ser sempre me ouvir e acalmar, por ser tão paciente e por me ajudar até de madrugada se preciso.

Deixo aqui também um agradecimento especial por pessoas que me ensinaram muito, não somente nas aulas, reuniões e trabalhos, mas principalmente na vida. Professora Dra. Lucia Rosana Moura, Professor Dr. Ricardo Felipe Custódio, minha amiga e Mestre Thaís Bardini Idalino e meus amigos e Mestres Felipe Sasso e Lucas Perin. Vocês participaram da experiência mais fantástica que vivi até hoje.

Por último, mas não menos importantes, agradeço aos meus colegas de trabalho por tornarem meus dias mais divertidos e por sempre conseguirem me fazer enxergar a correria dos últimos semestres de maneira leve.

Qualquer coisa é possível se você tiver coragem.

J.K.Rowling

RESUMO

É de comum conhecimento, dentro da área de Segurança em Computação, que dispomos de diferentes mecanismos para garantir que os princípios relacionados à mesma sejam garantidos. São eles: confidencialidade, integridade, disponibilidade e autenticidade. Um dos mecanismos conhecidos hoje chama-se Crypto Puzzle, ou Quebra-Cabeça Criptográfico. Trata-se de um problema matemático que deve ser resolvido para obter acesso a alguma coisa, seja ela uma informação básica ou até acesso aos diferentes serviços de um servidor. A utilização de quebra-cabeças criptográficos pode reforçar diferentes aspectos de segurança, desde garantir confidencialidade de uma informação por determinado tempo, até melhorar a disponibilidade de um serviço, servindo de mecanismo de controle de requisições contra ataques de negação de serviço (conhecidos como *DoS - Denial of Service*).

O presente trabalho pretende explorar os diferentes tipos de quebra-cabeça existentes, bem como a diferença, as vantagens e desvantagens entre eles e, baseado nas propriedades que cada um atende, irá mostrar a aplicabilidade de cada um. Além disso, este trabalho visa detalhar e implementar três abordagens diferentes de quebra-cabeças criptográficos, conhecidas como *Time Lock*, *Subset Sum* e *Modular Square Roots*. Estas três abordagens foram selecionadas por possuírem a propriedade de não paralelização, sendo úteis em cenários onde o tempo de resolução do quebra-cabeça é extremamente importante.

Além de detalhar e implementar estas três abordagens, uma série de experimentos será realizada em cada uma delas. Os resultados experimentais encontrados nos permitirão confirmar a eficiência das abordagens e compreender melhor os conceitos matemáticos envolvidos. Além disso, será possível comparar uma abordagem com a outra no que diz respeito à sua complexidade, custo computacional e precisão de tempo.

Palavras-chave: Quebra Cabeça Criptográfico. Segurança em Computação.

ABSTRACT

The information security and secrecy scenario can be explored in several ways. It is common knowledge, within the area of Computer Security, that we have different mechanisms to ensure that the principles related to it are guaranteed. These are: confidentiality, integrity, availability and authenticity. One of the mechanisms known today is called Cryptographic Puzzle. This is a mathematical problem that must be solved to gain access to something, be it basic information or even access to the different services of a server. The use of cryptographic puzzles can reinforce different aspects of security, from guaranteeing confidentiality of information for a certain time, to improving the availability of a service, serving as a mechanism to control requests against Denial of Service attacks (also known as DoS attacks).

The present work intends to explore the different types of puzzle, as well as the difference, the advantages and disadvantages between them and, based on the properties that each one attends, will show the applicability of each one. In addition, this work aims to implement three different approaches of cryptographic puzzles, known as Time Lock, Subset Sum and Modular Square Roots. These three approaches were selected because they have the non-parallelization property, which make them useful in scenarios where the puzzle' solving time is extremely important.

In addition to detailing and implementing these three approaches, a series of experiments will be conducted on each of them. Experimental results will allow us to confirm the efficiency of the approaches and to better understand the mathematical concepts involved. In addition, it will be possible to compare one approach with the other, considering its complexity, computational cost, and time precision.

Keywords: Cryptographic Puzzle. Computer Security.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O cenário de segurança atual fornece aos seus usuários diferentes maneiras de proteger-se contra ataques de pessoas maliciosas. Os ataques sofridos são de diferentes naturezas e têm diferentes objetivos, desde roubar senhas dos usuários e visualizar documentos aos quais não se deveria ter acesso, até inundar um servidor de requisições para tirar seus serviços do ar, afetando a disponibilidade do mesmo (STUTTARD; PINTO, 2011). Com base na gravidade dos problemas que estes ataques podem causar, ao lidar com informações sigilosas e grandes aplicações, é necessário que se faça uso de mecanismos de segurança, sendo possível optar pelo uso de mais de um no mesmo contexto, com o objetivo de garantir que as propriedades de segurança exigidas sejam garantidas.

Uma das maneiras de proteger-se contra diferentes tipos de ataque chama-se Quebra-Cabeça Criptográfico, que pode ser visto como um problema que deve ser resolvido pelo usuário para poder obter acesso à alguma coisa. A primeira menção de um *Crypto Puzzle* foi feita por Ralph Merkle em seu trabalho *Secure communications over insecure channels* (MERKLE, 1978). Segundo o autor, um quebra-cabeça pode ser definido como um criptograma que foi feito com o objetivo de ser quebrado. Assim como é possível cifrar um texto ao se produzir um criptograma com esses texto, deve ser possível proteger uma informação produzindo um quebra-cabeças. A diferença entre os dois conceitos é que um criptograma ideal não deve ser quebrado, enquanto o objetivo de um quebra-cabeças é, justamente, ser quebrado depois de um determinado esforço ou tempo.

Os quebra-cabeças criptográficos também são utilizados como proposta para resolver problemas gerados por ataques de negação de serviço, onde atacantes maliciosos enviam muitas requisições a um servidor e este, sem conseguir processar todas, fica indisponível para usuários legítimos. O objetivo, ao utilizar quebra-cabeças neste contexto, é fazer o cliente gastar uma grande quantidade de recursos, resolvendo o problema matemático proposto, antes de poder fazer requisições ao servidor, desta maneira um atacante terá que resolver um grande número de problemas matemáticos para conseguir inundar o servidor com requisições e, assim, realizar um ataque de negação de serviço (WATERS et al., 2004).

Dito isto, é possível perceber que este tipo de mecanismo de segurança pode ser aplicado em diferentes contextos. A aplicação mais conhecida é em cenários onde trabalha-se com o tempo. Para estes casos, o quebra-cabeça só pode ser quebrado depois de uma certa quantia de tempo, garantindo o sigilo da informação até o momento desejado (RIVEST; SHAMIR; WAGNER, 1996). Para atender à este contexto, o quebra-cabeça criptográfico deve possuir uma propriedade chamada "não paralelização". Esta característica diz que o cliente (como chamamos a pessoa que quer solucionar o problema proposto) não deve ser capaz de acelerar a resolução do problema em questão através da distribuição do mesmo em vários processadores (TRITILANUNT et al., 2007). Esta propriedade garante que o problema não poderá ser resolvido antes do tempo previsto, fazendo com que seja um mecanismo extra de segurança para casos onde queremos que uma informação apenas se torne disponível após um certo período de tempo.

Estes quebra-cabeças criptográficos não paralelizáveis são aplicados em diversos contextos. Em seu trabalho *Weakly Secret Bit Commitment: Applications to Lotteries and Fair Exchange* (SYVERSON, 1998), Paul Syverson fala sobre a utilização de quebra-cabeças para proteger o resultado de jogos de loteria. Neste cenário o resultado da loteria estaria basedo na lista de *tickets* vendidos, onde cada novo *ticket* vendido impactaria no resultado final da loteria. Uma vez encerrada as vendas, o tempo para computar o resultado da loteria deve ser suficientemente longo para que a lista final de *tickets* vendidos tenha sido publicada, para auditoria do resultado. Ao mesmo tempo deve ser suficientemente rápido para que ninguém possa calcular a saída da loteria antes do resultado oficial, através da utilização da lista publicada.

Tendo em mente tudo que foi exposto acima, os riscos existentes de sofrer um ataque malicioso e os problemas graves que isto pode causar em um sistema, bem como a importância de nos protegermos contra tais atacantes, observou-se a importância de estudar este mecanismo de segurança em computação na qual o presente trabalho se baseia. Quebras-cabeças criptográficos são mecanismos não estudados durante todo o curso de graduação de Sistemas de Informação e Ciência da Computação, e foi com base na falta deste conteúdo que este trabalho de conclusão de curso foi pensado, objetivando explicar detalhadamente as diferentes abordagens de quebra-cabeça, de forma a produzir um material com conteúdo atual e rico em informações e exemplos dentro do contexto tratado.

1.1 OBJETIVOS

A proposta deste trabalho não se atém somente ao estudo e detalhamento dos conceitos aqui tratados e à prática dos mesmos, como também objetiva o aumento do conhecimento de maneira geral sobre os aspectos mais importantes da segurança dentro do cenário atual no qual estamos imersos. Além disso, objetiva-se também reforçar conceitos vistos durante todo o curso de Sistemas de Informação, desde a programação mais básica, até os ensinamentos ligados ao planejamento de projetos e pesquisa.

Pretende-se também expandir o campo de pesquisa para aprender novos assuntos que não foram cobertos durante o curso, mas que já são de possível entendimento devido à base fornecida durante os últimos anos. Para aumentar o conhecimento, uma nova linguagem foi estudada para implementar os algoritmos escolhidos, devido à abordagem funcional e rapidez com que a mesma opera. Além disso, foi necessário pesquisar, estudar, entender e exercitar diversos conceitos matemáticos para compreender conceitos utilizados nos problemas analisados.

Abaixo serão apresentados os objetivos desta pesquisa de maneira mais técnica e específica.

1.1.1 Objetivo geral

O foco deste trabalho está em compreender o cenário atual de quebra-cabeças criptográficos, conseguindo diferenciar cada uma das propostas existentes de acordo com a melhor aplicação que lhes cabe. Pretende-se fazer um estudo sobre cada uma delas, suas propriedades, vantagens e desvantagens, e expor a pesquisa de maneira que fiquem claras as diferentes abordagens existentes. Além disso, este trabalho visa explicar detalhadamente três abordagens específicas que trabalham com o fator tempo, e implementá-las para a realização de uma série de testes, expondo também os resultados experimentais encontrados.

1.1.2 Objetivos específicos

Os objetivos específicos deste trabalho são elencados abaixo.

- Explicar o cenário atual de quebra-cabeças criptográficos;

- Fornecer um panorama geral de algumas importantes abordagens existentes e sua aplicabilidade;
- Explicar detalhadamente o funcionamento das três abordagens de quebra-cabeças criptográficos relacionadas ao tempo;
- Implementar as três abordagens de quebra-cabeças criptográficos atreladas ao tempo, onde para cada uma deve-se:
  - Implementar método de criação do quebra-cabeça;
  - Implementar método de solução do quebra-cabeça;
  - Implementar método de verificação da solução;
  - Produzir e rodar testes;
  - Comparar e estudar os resultados experimentais;
- Como resultado do trabalho, produzir um texto que sirva como material para estudo sobre quebra-cabeças criptográficos, suprindo a falta deste conteúdo nos cursos de Sistemas de Informação e Ciência da Computação;

1.2 JUSTIFICATIVA E MOTIVAÇÃO

A segurança e o sigilo das informações que circulam pela internet nos dias de hoje baseia-se em diferentes mecanismos. A importância de existir diferentes formas de proteger-se é evidente quando sabemos a quantidade de ataques que pessoas mal intencionadas podem realizar e, principalmente, quando nos damos conta das consequências que decorrem de tais atos. Um dos mecanismos conhecidos na área de segurança em computação é o quebra-cabeça criptográfico, objeto de estudo deste trabalho.

Existem hoje algumas implementações de quebra-cabeças criptográficos, contemplando as mais variadas características e, portanto, os mais variados contextos em que podem ser utilizados, sendo úteis em diferentes aspectos da segurança. Tendo em vista a importância deste mecanismo, bem como a vasta aplicabilidade dele, este trabalho propõe o estudo de diferentes abordagens de quebra-cabeça existentes, reforçando o entendimento dos conceitos envolvidos, e produzindo uma implementação que teste as principais abordagens estudadas, de maneira que seja possível obter resultados experimentais em cada uma delas.

1.3 METODOLOGIA

O desenvolvimento deste trabalho se iniciou com a pesquisa do estado da arte de quebra-cabeças criptográficos, com enfoque naqueles que possuem a propriedade de "não paralelização"(que trabalham com o fator tempo), como veremos no Capítulo 3. Os estudos, porém, não se ativeram somente a este tipo específico de quebra-cabeça, como também a diversas outras abordagens de quebra-cabeça, separando cada uma de acordo com suas características e aplicabilidade.

A segunda fase tratou de selecionar alguns quebra-cabeças importantes de serem estudados, compreendendo assim cada detalhe das três fases de implementação de cada um deles: construção, solução e verificação do problema matemático. A escolha foi tomada com base nas propriedades que cada um atende, focando em implementar aqueles que possuem a propriedade de "não paralelização", conforme será explicado no capítulo 3. Uma vez escolhidas as abordganes, foram feitos grupos de estudo a respeito dos conceitos matemáticos utilizados em cada uma delas e, após o entendimento dos mesmos, foi possível iniciar o desenvolvimento dos algoritmos que implementam os quebra-cabeças. Cada uma das três abordagens escolhidas para implementação será detalhada no Capítulo 4.

Para decidir que linguagem utilizar, foi feita uma pesquisa entre as mais populares para verificar quais têm as características desejáveis para este tipo de programa. Um ponto decisivo na escolha foi a quantidade de bibliotecas que poderiam auxiliar e agilizar o trabalho quando se tratava de conceitos matemáticos complexos, como o algoritmo de redução de bases de reticulado de Lenstra–Lenstra–Lovász (também conhecido como LLL ou L^3) (LENSTRA; LENSTRA; LOVÁSZ, 1982), como veremos no Capítulo 2. A linguagem escolhida, de acordo com o que foi exposto acima, foi Python (PYTHON, 2001) em sua versão 3.5.

Na implementação contamos com métodos para a construção do quebra-cabeça, fase em que o problema matemático a ser resolvido é estruturado, produzindo o quebra-cabeça em si, que deve retornar, além do problema em questão, os parâmetros necessários para a solução do mesmo. Contamos também com métodos para a resolução do problema proposto, que devem retornar a solução encontrada e, em alguns casos, credenciais específicas da comunicação entre as partes envolvidas no processo. Por fim, foram criados também métodos para a verificação

da solução encontrada, garantindo a corretude dos métodos de solução.

Para testar os algoritmos implementados, foi esquematizada uma rotina de testes unitários onde, para cada algoritmo, seriam rodados alguns testes. Esta rotina testa a criação do quebra-cabeça, cifrando junto uma informação qualquer, que deve ser decifrada após a solução do problema matemático, também implementada e inserida na rotina de testes. Por fim, o código foi avaliado para assegurar o correto funcionamento e segurança das rotinas implementadas. Cumpridos os objetivos de implementação do programa, três mecanismos de quebra-cabeça criptográfico distintos foram implementados e testados, e seus resultados foram expostos em um capítulo de resultados experimentais.

1.4 ESTRUTURA DO DOCUMENTO

No Capítulo 2 são apresentados conceitos teóricos relacionados aos problemas matemáticos, cujo entendimento se faz fundamental para a compreensão dos mecanismos que serão detalhados futuramente. No Capítulo 3 é abordada a introdução aos quebra-cabeças criptográficos existentes, bem como as propriedades que cada um deles pode ter. Para cada abordagem estudada será evidenciado o contexto em que é aplicável dentro da segurança em computação.

Já no Capítulo 2, focalizamos em três abordagens específicas de quebra-cabeça criptográfico que têm a propriedade de "não paralelização", ou seja: é aqui que detalhamos os quebra-cabeças que trabalham com o fator tempo. Para cada uma das três abordagens são explicadas em detalhes as fases do problema em questão (são elas: criação, solução e verificação). Nos Capítulos 5 e 6 são mostrados todos os aspectos relacionados à implementação destas três abordagens, incluindo as especificações de linguagem e bibliotecas utilizadas.

No Capítulo 7 são apresentados os resultados experimentais obtidos através da criação e execução de testes unitários. Por fim, com base no que foi estudado e nos testes executados, o Capítulo 8 apresenta a conclusão e as idéias para trabalhos futuros, visto que há muitas possibilidades de como estender esta pesquisa.

No final deste documento encontram-se as referências e os anexos, onde é possível verificar o código criado para cada abordagem,

bem como o artigo produzido com base nesse trabalho, salientando apenas as partes fundamentais para o entendimento do que foi feito, e da importância do mesmo.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta três conceitos que são de extrema importância para o entendimento das abordagens de quebra-cabeça criptográficos que serão apresentadas, implementadas e testadas. Tais conceitos são: a função *hash*, na Seção 2.1, o problema da mochila, na Seção 2.2, o algoritmo de Lenstra-Lenstra-Lovász (LENSTRA; LENSTRA; LOVÁSZ, 1982), na Seção 2.3, conceitos matemáticos envolvendo resíduos quadráticos e não quadráticos, conforme exposto na Seção 2.4 e, por fim, as definições de complexidade (linear, polinomial e exponencial) na Seção 2.5.

2.1 FUNÇÃO HASH

Uma função de *hash* criptográfica h mapeia uma entrada M de tamanho variável de cadeia para uma *string* (texto) de comprimento fixo em *bits*. As funções de *hash* criptográficas têm muitas aplicações, como: são utilizadas em assinaturas digitais, carimbo do tempo (*timestamp*) e como método de detecção de modificação de arquivos (RIVEST, 2008).

Para ser usada no contexto de segurança, a função *hash* deve contemplar algumas propriedades:

- Resistência a pré-imagem: deve ser inviável para um adversário, dado y, calcular M, tal que $hash(M) = y$;

- Resistência à colisão: deve ser inviável para um adversário encontrar valores M_1 e M_2, tal que $hash(M_1) = hash(M_2)$;

- Resistência à segunda pré-imagem: deve ser inviável para um adversário, dado M_1, encontrar um valor diferente M_2, tal que $hash(M_1) = hash(M_2)$;

As funções de *hash* da família SHA-2 (National Institute of Standards and Technology (NIST), 2015) serão utilizadas em todas as abordagens estudadas neste trabalho.

2.2 PROBLEMA DA MOCHILA

Trata-se de um problema que tem como objetivo encontrar um subconjunto de valores, dado um conjunto de números nomeados "pesos", cuja soma resulte no maior número possível, desde que este não ultrapasse um valor S previamente estipulado (TRITILANUNT et al., 2007). Podemos pensar em S como sendo a capacidade total de peso que uma mochila pode carregar, e os diferentes números do conjunto recebido podem ser considerados itens, cuja soma de uma coleção destes itens não deve exceder a capacidade máxima da mochila, por isto o problema recebeu este nome.

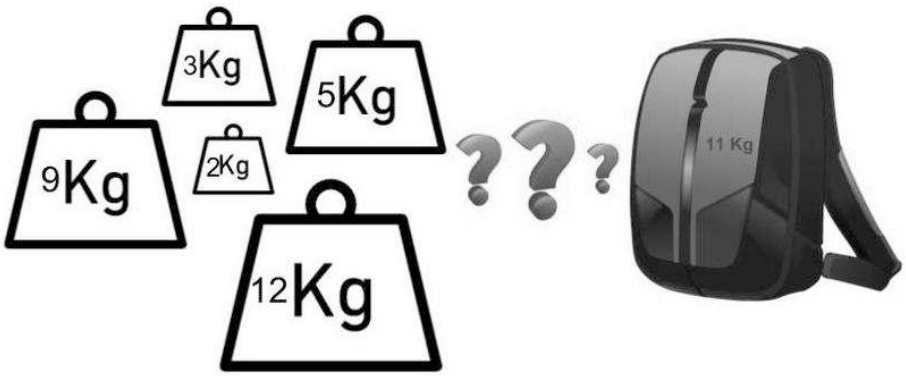

Figura 1 – Problema da Mochila

Para melhor entendimento, podemos pensar no caso da Figura 1, onde o melhor subconjunto para o peso estipulado na mochila seria composto pelos itens $\{2, 9\}$. Neste contexto fica muito fácil encontrar a resposta, porém este problema, quando aplicado em cenários de segurança, tende a ser muito difícil por estipular um valor alto como peso máximo, disponibilizando um conjunto enorme de itens, de maneira que fica difícil decidir qual o melhor subconjunto rapidamente, apenas visualizando os itens disponibilizados.

O problema da mochila é conhecido por ser NP-completo e, portanto, bastante complexo, visto que os problemas NP-completo são os mais difíceis dentro de uma classe famosa chamada NP. Ser NP-completo significa que, caso fosse possível encontrar uma maneira de

resolver este problema NP-completo rapidamente (em tempo polinomial), então poderiam ser encontrados algoritmos para resolver também todos os outros problemas da classe NP rapidamente (TRITILANUNT et al., 2007). Dada a dificuldade do problema, um dos mecanismos para solucioná-lo é o chamado algoritmo LLL, ou L^3, como veremos a seguir.

2.3 ALGORITMO *LLL*

Antes de falarmos do algoritmo de Lenstra-Lenstra-Lovász (LENSTRA; LENSTRA; LOVÁSZ, 1982), é necessário introduzir o conceito de reticulados. Em matemática, um reticulado é uma estrutura $L = (L, R)$ tal que L tem seus elementos parcialmente ordenados por uma relação R e, para cada dois elementos a, b no conjunto L, existe supremo (menor limite superior) e ínfimo (maior limite inferior) de a, b (DIAS, 2013).

Para prosseguir, é preciso também entender o conceito de vetores linearmente independentes. Seja V um espaço vetorial e $v_1, v_2, v_3, ..., v_n \in V$ alguns de seus vetores, considerando a equação:

$$a_1v_1 + a_2v_2 + a_3v_3 + ... + a_nv_n = 0 \tag{2.1}$$

Se a única solução possível para esta equação for

$$a_1 = a_2 = a_3 = ... = a_n = 0$$

então dizemos que os vetores de V são Linearmente Independentes. Já se houver mais de uma solução para esta equação, dizemos que $v_1, v_2, v_3, ..., v_n$ são Linearmente Dependentes. Quando estamos tratando de reticulados, este conceito é necessário para entender a definição de base.

Se pensarmos em $m \leq n$ vetores linearmente independentes: $v_1, v_2, ..., v_m \in \mathbb{R}^n$, um reticulado Λ com base $(v_1, v_2, ..., v_m)$ é o conjunto de todas as combinações lineares inteiras de v_i, onde $1 \leq i \leq m$ (CAMPELLO, 2014), o que significa:

$$\Lambda = \{u_1v_1 + ... + u_mv_m : u_1, ..., u_m \in \mathbb{Z}\} \tag{2.2}$$

O conjunto $\{v_1, v_2, ..., v_m\}$ então é denominado uma base de Λ.

Sabendo os conceitos definidos acima, é possível entender para que serve e como funciona o algoritmo *LLL*, também conhecido como L^3. O uso de reticulados é frequente dentro da matemática e da criptografia, sendo o problema do vetor mais curto (conhecido como *SVP - Shortest vector problem*) o problema mais famoso e mais estudado. Ele consiste em encontrar o vetor mais curto dentro de um reticulado (XINYUE, 2016). Este é o problema que deverá ser resolvido em uma das abordagens de quebra-cabeças implementadas neste trabalho.

Para encontrar o vetor mais curto é necessário fazer uma redução de base, que é um processo utilizado para reduzir uma base B de um reticulado, substituindo-a por uma base B' com vetores de menor tamanho (norma), sem que isso mude o reticulado. Utilizamos o algoritmo de Lenstra-Lenstra-Lovász para fazer esta redução de maneira eficiente. Este algoritmo é utilizado pois, segundo Tritilanunt et al. (2007), é o método mais eficiente conhecido para encontrar vetores pequenos. O processo de construção das bases, bem como a utilidade do vetor mais curto dentro do processo de solução de um *Subset Sum Puzzle* é melhor descrito no Capítulo 4, ao explicar a fase de solução do protocolo.

Para o melhor entendimento de cada um dos passos que o algoritmo *LLL* executa para encontrar o vetor mais curto, é possível consultar o fluxo e os pontos de decisão do algoritmo nos trabalhos *Toward non-parallelizable client puzzles* (TRITILANUNT et al., 2007) e *Solving Subset Sum Problems with L3 Algorithm* (RADZISZOWSKI; KREHER, 1988).

2.4 RESÍDUOS

Para entender os conceitos relacionados a resíduos quadráticos e não quadráticos, é necessário fazer uma revisão de como extrair raízes quadradas em módulo primo. Se tivermos um número primo p, e um número inteiro a em $\mathbb{Z}_p^*$, onde $1 \leq a \leq p-1$, então a solução da congruência $x^2 \equiv a \pmod{p}$ é chamada de raiz quadrada em módulo p. A solução pode ser inexistente ou, se existir, serão duas soluções x e $-x$. No primeiro caso, onde existe solução, a é chamado de resíduo não quadrático, enquanto no segundo, a é chamado de resíduo quadrático em módulo p (JERSCHOW; MAUVE, 2011). Um resíduo quadrático é um elemento que é equivalente ao quadrado de algum outro elemento em $\mathbb{Z}_p^*$ (BARROS, 2008).

Por exemplo, em $\mathbb{Z}_7^*$:

$$1^2 \equiv 1 \pmod 7$$
$$2^2 \equiv 4 \pmod 7$$
$$3^2 \equiv 9 \equiv 2 \pmod 7$$
$$4^2 \equiv 16 \equiv 2 \pmod 7$$
$$5^2 \equiv 25 \equiv 4 \pmod 7$$
$$6^2 \equiv 36 \equiv 1 \pmod 7$$

É possível afirmar, através das equações acima, que 1, 2 e 4 são resíduos quadráticos em $\mathbb{Z}_7^*$, enquanto os outros elementos do conjunto $\{3, 5, 6\}$ são resíduos não quadráticos em módulo 7, pois não são equivalentes ao quadrado de nenhum outro elemento do conjunto. Para dizer se a é, ou não, um resíduo quadrático, é utilizado o símbolo de Legendre, denotado como $\left(\frac{a}{p}\right)$. O símbolo de Legendre é uma função cujo valor é 1 se a é um resíduo quadrático módulo p, -1 se é um resíduo não quadrático módulo p e 0 se $a = 0$.

2.5 COMPLEXIDADE

Os algoritmos que serão detalhados e testados no decorrer do presente trabalho serão analisados de acordo com uma série de propriedades, sendo uma delas a complexidade de cada um deles, relacionada ao tempo que cada um leva para realizar as operações exigidas na respectiva abordagem. Para os mecanismos de quebra-cabeças criptográficos estudados, precisamos entender três medidas diferentes de complexidade:

- Complexidade linear: melhor caso possível, onde o tempo de execução (medido como *O(n)*) aumenta linearmente de acordo com o tamanho da entrada, possibilitando um controle muito pontual do tempo de resolução do problema matemático envolvido;
- Complexidade polinomial: onde o tempo de execução é delimitado por uma expressão polinomial ($T(n) = O(n^k)$) para um número

k fixo. Embora um tempo linear também possa ser polinomial, neste contexto temos a certeza de que, em casos onde o tempo é polinomial, não há um controle pontual do tempo de resolução do problema, pois as saídas variam de maneira mais abrupta dependendo da entrada fornecida;

- Complexidade exponencial: onde o tempo de execução é delimitado por ($T(n) = O(a^n)$) para $a > 1$ fixo. Neste caso o controle do tempo de resolução também não é pontual;

3 VISÃO GERAL DE QUEBRA-CABEÇAS CRIPTOGRÁFICOS

3.1 INTRODUÇÃO

Os quebra-cabeças criptográficos foram inicialmente propostos em 1978 por Ralph Merkle. Em seu trabalho, Merkle utilizou os quebra-cabeças para discutir conceitos básicos relacionados à comunicação segura em canais inseguros (MERKLE, 1978). Desde então os quebra-cabeças criptográficos tem sido utilizados em diversas aplicações, como por exemplo, uma técnica para enviar um documento para o futuro ou para dificultar o ataque de negação de serviço em redes de computadores.

Existem diferentes propriedades que um quebra-cabeça criptográfico pode atender, e a obrigatoriedade destas vai depender do contexto em que está sendo aplicado.Tais propriedades serão revisadas e detalhadas ainda neste capítulo.

Além das características desejáveis, outra questão que começou a receber foco foi a interatividade do servidor (parte que gera o quebra-cabeça) com o cliente (parte que resolve o quebra-cabeça). Quando um problema é interativo (situação que ocorre em quase todas as abordagens conhecidas até agora), o servidor envia o problema para o cliente junto com os parâmetros necessários para a solução do mesmo. Porém, pode haver um atacante no meio desta comunicação que consiga alterar os parâmetros, fazendo com que o cliente não consiga solucionar o problema proposto. Tendo em vista este problema, uma nova arquitetura de quebra-cabeça foi criada em *Non-parallelizable and non-interactive client puzzles from modular square roots* (JERSCHOW; MAUVE, 2011), possibilitando que o cliente e o servidor não interajam nas fases de criação e solução. Esta arquitetura também será abordada neste trabalho.

Quebra-cabeças criptográficos são muito utilizados em diversas áreas da segurança computacional. Alguns exemplos do uso desses quebra-cabeças são: aplicações de loteria, onde o resultado dos jogos só pode ser publicado depois de um certo tempo (SYVERSON, 1998) e envio de documentos após a data limite de entrega, chamado *offline submission*, ou submissão de documentos *offline* (JERSCHOW; MAUVE, 2010). Nesse caso, um indivíduo precisa enviar um documento até

um certo prazo, mas não consegue devido à problemas técnicos, como perda de conexão com Wi-fi. Ele então cria um quebra-cabeça com o *hash* do seu documento, e envia quando recuperar a conexão, comprovando assim que o documento que gerou aquele *hash* já estava pronto na data estipulada (horário em que o quebra-cabeça foi gerado). Pode ser utilizado também para assinatura de contratos digitais (GARAY; POMERANCE, 2003), onde as duas partes que querem assinar o contrato precisam confiar uma na outra para assinar simultaneamente um documento. Para isto, ambas concordam em revelar um segredo gradualmente, à medida que ganham conhecimento sobre o segredo da outra parte. Pode ainda ser utilizado para evitar ataques DoS (WATERS et al., 2004), entre muitas outras aplicações.

3.2 PROPRIEDADES

Para o seu pleno uso, quebra-cabeças criptográficos devem possuir uma série de propriedades, como veremos a seguir. É difícil conseguir alcançar todas as características existentes, portanto é importante ter em mente qual a finalidade de utilizar um mecanismo desses, pois assim fica mais claro quais dos requisitos desejáveis são realmente indispensáveis para uma aplicação em específico, criando assim uma cadeia de priorização destas características.

Para melhor entendermos estas propriedades, é necessário definir os conceitos de servidor e cliente. Dá-se o nome de servidor à parte que constrói o quebra-cabeça. Dá-se o nome de cliente à parte que irá solucionar o quebra-cabeça. Algumas das propriedades desejáveis em um quebra-cabeça foram estudadas neste trabalho, por serem as mais interessantes dentro do contexto em que os mesmos podem ser aplicados. Outras propriedades e critérios analisados em quebra-cabeças criptográficos podem ser analisados em *DOS-resistant Authentication with Client Puzzles* (AURA; NIKANDER; LEIWO, 2000). As propriedades aqui estudadas são:

- Unidade de Trabalho: Tipo das operações requeridas ao cliente para a solução do problema (por exemplo: operações de quadrado modular);

- Custo do Servidor: Esforço computacional da parte que cria o quebra-cabeça, medindo o custo em todas as fases do problema

que dizem respeito ao servidor, desde sua pré-computação (nos casos em que houver), construção e verificação do problema;

- Custo do Cliente: Esforço computacional da parte que resolve o quebra-cabeça, na hora de receber, solucionar e enviar a resposta de volta ao servidor. Assume-se que cliente e servidor tenham capacidade de processamento e memória semelhantes. Para algumas abordagens, como veremos, o cliente também terá custo de construção;

- Não Paralelização: Capacidade de evitar que a parte que irá solucionar o quebra-cabeça possa fazê-lo de maneira mais rápida através da paralelização do trabalho em diferentes máquinas, fazendo assim com que a abordagem deixe de ser aplicável em contextos envolvendo um tempo específico para solução;

- Complexidade: Complexidade de cada algoritmo, com base no tempo, conforme visto na Seção de Fundamentação Teórica;

3.3 QUEBRA-CABEÇAS CRIPTOGRÁFICOS

Abaixo será feita uma breve revisão das diferentes abordagens de quebra-cabeças criptográficos estudadas, revisando o conceito, a aplicabilidade, as propriedades atendidas por cada uma delas, bem como as vantagens e desvantagens de cada uma. Vale lembrar que as vantagens e desvantagens também vão variar dependendo do contexto em que será utilizada.

Das abordagens estudadas, foram selecionadas três que possuíam a propriedade de "não paralelização", desejável em cenários relacionados com o tempo. Estas três foram escolhidas para serem implementadas e testadas neste trabalho e serão descritas em detalhes no Capítulo 4. É importante ressaltar que estas não são as únicas abordagens que contemplam a propriedade de "não paralelização", mas foram selecionadas por serem amplamente conhecidas dentro da área de segurança da computação. O código correspondente a cada uma delas será apresentado no capítulo 6.

3.3.1 Hash-based Reversal Puzzles

Nesta abordagem, o problema proposto é que o cliente encontre os bits que estão faltando em uma pré-imagem da função *hash* enviada pelo servidor. Ao enviar esta pré-imagem, envia também o *hash* completo (sem os bits faltantes). Desta maneira, o servidor pode ajustar o nível de dificuldade através do aumento ou diminuição do número de bits faltantes na pré-imagem enviada como quebra-cabeça, quanto mais bits estiverem faltando, mais difícil será para resolvê-lo. O cliente então, ao receber a pré-imagem do servidor, deve realizar força bruta para achar o padrão que resulta nos bits faltantes. Para isto, necessita gerar um *hash* para cada tentativa, verificando se este é idêntico ao *hash* completo enviado pelo servidor junto ao quebra-cabeça (BRAINARD; JUELS, 1999).

Quando o cliente conseguir encontrar o valor *hash* igual ao enviado pelo servidor, ele envia a resposta para o mesmo. Dessa maneira, o servidor faz apenas uma operação *hash* em todo o processo, sendo assim fácil de construir e verificar, porém com complexidade exponencial (onde o tempo de execução é delimitado por ($T(n) = O(2^{n^k})$) para qualquer entrada k). Esta abordagem é paralelizável pelo fato de ser solucionada através de força bruta, ou seja: diferentes máquinas podem rodar o mesmo quebra-cabeça ao mesmo tempo e uma pode encontrar a solução muito antes que outra, pois estarão testando diferentes possibilidades. O detalhamento de cada etapa, bem como o fluxo completo do protocolo para cada parte envolvida podem ser observados no trabalho *Client Puzzles: A Cryptographic Defense Against Connection Depletion Attacks*, de Brainard e Juels (1999).

3.3.2 Hint-based Hash Reversal Puzzles

Esta abordagem é um aprimoramento do método anterior, porém com algumas melhorias que permitem que a complexidade seja linear, ao invés de exponencial. Neste caso, o servidor vai fornecer ao cliente informações extras, como se fossem pistas, anexadas ao quebra-cabeça. Desta maneira, ao invés de checar todas as soluções possíveis, o cliente testa somente aquelas que estão dentro do conjunto de pistas fornecidas, diminuindo muito o campo de busca pela solução do problema (FENG et al., 2005).

Contudo, esta abordagem ainda não fornece a propriedade de "não paralelização", justamente por existir este conjunto de pistas, pois, se colocarmos diversas máquinas tentando descobrir a solução, dando a elas um conjunto de possíveis soluções, uma pode encontrar a reposta correta muito antes da outra, visto que cada uma poderá começar testando uma solução diferente dentro do conjunto de pistas.

3.3.3 DH-based Puzzles

Esta abordagem proposta por Waters et al. (2004) leva este nome por ser baseada no algoritmo de Diffie-Hellman, amplamente conhecido na área de segurança da computação. A troca de chaves de Diffie-Hellman é um método para troca de chaves criptográficas desenvolvido por Whitfield Diffie e Martin Hellman (DIFFIE; HELLMAN, 1976). Este método permite que duas partes entrem em acordo e compartilhem uma chave secreta de forma segura sob um canal de comunicação inseguro. Para entender essa abordagem, é necessário entender como este protocolo funciona.

As duas partes interessadas em trocar chaves devem escolher dois valores p e g que podem ser públicos. Estes valores vão representar um número primo e uma base. Cada uma das partes vai escolher também um valor que manterá em segredo, que irá ser utilizado no processo de geração da chave secreta. Para entendermos melhor, podemos pensar nos valores $p = 97$ e $g = 5$.

A primeira parte, Alice, escolhe como valor secreto $a = 36$, enquanto a segunda parte, Bob escolhe $b = 58$. Primeiro Alice irá utilizar os valores previamente compartilhados para computar

$$V_a = 5^{36} \mod 97 = 50$$

enquanto Bob, seguindo o mesmo padrão, irá computar

$$V_b = 5^{58} \mod 97 = 44$$

.

Ambos enviam os valores encontrados um para o outro. Agora Alice utilizará o valor recebido para computar

$$K = V_b^{36} \mod 97 = 44^{36} \mod 97 = 75$$

enquanto Bob, seguindo o mesmo padrão, irá computar

$$K = V_a^{58} \mod 97 = 50^{58} \mod 97 = 75$$

.

Ambos chegaram no mesmo valor $K = 75$, que agora passa a ser a chave secreta compartilhada por eles.

A construção desse quebra-cabeça requer uma operação de Diffie-Hellman, que vai resultar em um valor $g_{c,r}$, visto como uma chave pública no acordo de chaves de Diffie-Hellman, gerado a partir da equação:

$$g_{c,r} = g^{f'(a_{c,T})} \mod p \tag{3.1}$$

onde f' é uma permutação e $a_{c,T}$ é um inteiro escolhido aleatoriamente. O servidor fornece, junto ao quebra-cabeça, um conjunto de possíveis valores de a para o cliente testar. Este, por sua vez, testa cada valor esperando satisfazer a igualdade:

$$g_{c,r} = g^{f'(a')} \mod p \tag{3.2}$$

A verificação é barata, pois o servidor já gera a solução do problema em tempo de construção e a deixa armazenada na memória, sendo assim necessário apenas uma consulta à tabela no banco para verificação da solução recebida.

Devido ao fato desta abordagem também fornecer um conjunto de possíveis soluções, esta também possui a desvantagem de ser paralelizável, pois cada máquina pode testar um valor diferente em paralelo para tentar encontrar mais rápido a solução correta dentro do conjunto de soluções possíveis fornecidas pelo servidor.

3.3.4 Trapdoor RSA-based Puzzles

Esta abordagem foi feita pensando na redução de trabalho na fase de construção do problema, pois alguns parâmetros e operações caras já estão pré-computados (GAO, 2005). Este protocolo computa e armazena as soluções em tempo de geração do problema, economizando a carga de trabalho também em tempo de verificação. O servidor então só precisa de uma comparação (através de uma consulta simples à

tabela que contém as soluções) para checar a validade da solução fornecida pelo cliente. Por conta disto a carga de trabalho pré-computacional é bastante desvantajosa, agregando uma fase a mais ao processo de geração do quebra-cabeça.

Como o próprio nome diz, esta abordagem é feita utilizando funções *trapdoor*. Estas funções são conhecidas por serem fáceis de computar em um sentido, porém difíceis de computar em sentido oposto, caso não haja nenhuma informação extra sobre a mesma. Porém, com esta informação extra fica fácil de computar. Em termos matemáticos, se tivermos uma função $f(x)$, é fácil computar x se tivermos uma informação secreta y. Esta informação seria o chamado alçapão *trapdoor*.

O cliente, ao receber um quebra-cabeça, vai receber uma série de possíveis soluções para executar força bruta e descobrir qual a correta para o problema matemático proposto enviado pelo servidor. Porém, assim como a abordagem de *Hint-based Hash Reversal Puzzles* vista anteriormente, ao fornecer um conjunto de possíveis soluções, a abordagem passa a ser paralelizável, pois é possível encontrar a solução correta mais facilmente distribuindo a busca pela mesma em diferentes máquinas.

3.3.5 Ma's Hash Chain Reversal Puzzles

As abordagens de quebra-cabeças que trabalham com cadeia de valores (do inglês *chain*) atendem à propriedade de "não paralelização", pois, por se tratar de uma cadeia, é necessário o valor de uma operação anterior para conseguir realizar a operação seguinte. Esta abordagem, proposta por Ma em *Mitigating denial of service attacks with password puzzles* (MA, 2005) trabalha com uma cadeia de operações *hash*. Uma cadeia linear de operações *hash* é construída seguindo o fluxo da Figura 2.

A construção deste problema começa com um número h_0 qualquer escolhido randomicamente, sobre o qual será aplicada uma função *hash* k vezes, sendo k o tamanho desejável da cadeia. Toda a cadeia produzida é armazenada na memória, tornando a fase de verificação bastante simples, pois não há nada novo a ser gerado para a conferência do resultado. Em contrapartida, existe um grande uso de memória neste protocolo.

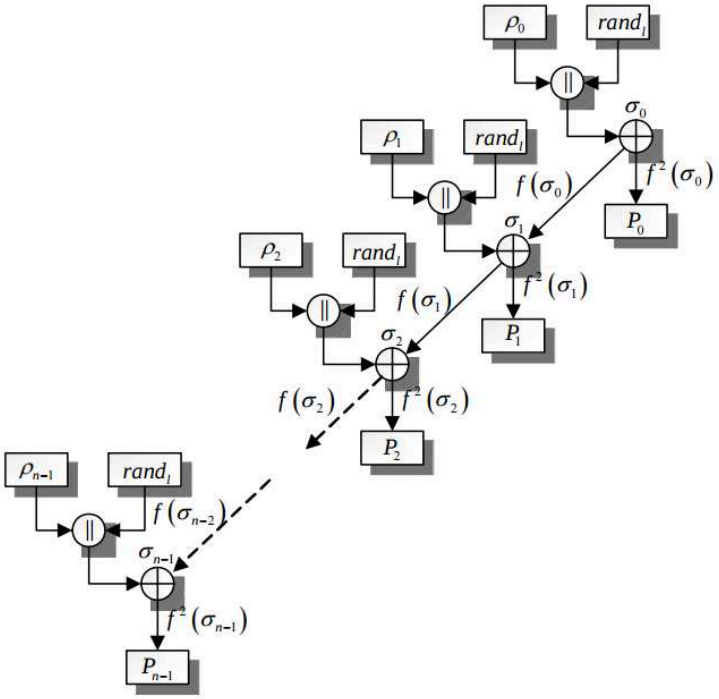

Figura 2 – Cadeia linear de *hash* (GROZA; PETRICA, 2006)

A cadeia, neste algoritmo, segue a seguinte lógica de criação:

$$h_{i+1} = hash(h_i) \tag{3.3}$$

onde $0 \leq i \leq k$. Para resolver o problema, o cliente recebe o último valor de *hash* produzido (h_k) junto com o valor de k, e tem que obter toda a cadeia de *hash* até chegar em h_0. Esta abordagem, porém, traz muitos problemas práticos, como a necessidade de uma grande capacidade de armazenamento para guardar toda a cadeia de *hash* para posterior verificação. Além disso, quando é utilizado uma função *hash* criptográfica comum, o caminho inverso pode ficar muito complicado de ser feito, pois esse tipo de função é difícil de reverter. Com o intuito de resolver este impasse, a próxima abordagem de quebra-cabeça foi criada.

3.3.6 Groza and Petrica's Hash Chain Puzzles

Segundo Groza e Petrica (2006), esta abordagem contorna o problema visto anteriormente quanto à dificuldade de fazer o caminho inverso em funções *hash* criptográficas comuns. Nesta abordagem, vol-

tada especificamente para cenários de negação de serviço, quanto mais etapas da cadeia de *hash* forem realizadas pelo cliente, mais acesso ele ganha aos diferentes serviços do servidor. Dessa maneira, o cliente não necessariamente ganha acesso a todos os serviços de uma só vez. O protocolo funciona da seguinte maneira: o servidor escolhe dois números aleatoriamente e os concatena para gerar σ_0. A primeira saída do problema (P_0), é gerada fazendo *hash* duas vezes do valor σ_0:

$$P_0 = hash(hash(\sigma_0)) \tag{3.4}$$

onde:

$$\sigma_0 = p||q \tag{3.5}$$

onde p e q representam os números escolhidos de forma aleatória. A partir daí, cada etapa da cadeia é criada fazendo XOR da variável σ_i da etapa anterior com a concatenação de dois novos números aleatórios:

$$P_{i+1} = \sigma_i \oplus \sigma_{i+1} \tag{3.6}$$

onde:

$$\sigma_{i+1} = p_{i+1}||q_{i+1} \tag{3.7}$$

O quebra-cabeça recebido pelo cliente é uma série de pares:

$$[(P_0, q_0), (P_1, q_1), ..., (P_n, q_n)] \tag{3.8}$$

onde n corresponde ao tamanho da cadeia, tal que $1 \leq n$. O cliente então precisa descobrir cada número p_i que é concatenado com o número q_i em cada etapa, encontrando o valor σ_i. Com esse valor, é possível fazer XOR com o valor P_i de cada etapa, encontrando o valor σ da etapa anterior.

Esta abordagem, porém, acaba exigindo muito esforço computacional por parte do servidor, tanto na fase de construção, quanto na fase de verificação, violando um requisito básico de quebra cabeças-criptográficos voltados para cenários de negação de serviço: O problema deve ser fácil de construir e verificar pelo servidor, mas difícil de solucionar para o cliente (TRITILANUNT et al., 2007).

4 ABORDAGENS DE QUEBRA-CABEÇAS CRIPTOGRÁFICOS NÃO PARALELIZÁVEIS

Este capítulo apresenta em detalhes três quebra-cabeças criptográficos estudados que possuem a propriedade de "não paralelização". Foram descritas as fases de contrução, de solução e de verificação do mesmo, exemplificando o modelo através de exemplos matemáticos. A Seção 4.1 descreve o quebra-cabeça proposto por Rivest, nomeado *Time Lock* (RIVEST; SHAMIR; WAGNER, 1996), que trabalha com operações de quadrado em módulo. A Seção 4.2 descreve o quebra-cabeça baseado na soma de subconjuntos. Por este motivo, recebeu o nome de *Subset Sum* (TRITILANUNT et al., 2007). Já a Seção 4.3 apresenta a técnica baseada na obtenção de raízes quadradas discretas, que recebe o nome de *Modular Square Roots* (JERSCHOW; MAUVE, 2011). O entendimento destas abordagens se faz necessário para compreender a implementação das mesmas, bem como os resultados dos testes experimentais que serão expostos no Capítulo 7.

4.1 TIME LOCK

A técnica Quadrados Repetidos ou *Time Lock Puzzle* (RIVEST; SHAMIR; WAGNER, 1996) foi uma das primeiras a serem criadas. Nesse mecanismo, o desempenho é medido pelo número de operações de quadrado modular que podem ser realizadas por uma máquina em um determinado período de tempo. Baseado nesse valor e dado um certo número de operações de quadrado a serem executadas, é possível estimar a quantidade de tempo necessária para resolver o quebra-cabeças.

Para resolver o quebra-cabeça, o cliente precisa computar repetidamente uma operação de quadrado modular de maneira sequencial, onde cada operação ocorre, necessariamente, uma após a outra. Isso faz com que não seja possível a paralelização deste quebra-cabeça, pois independente de quantas máquinas sejam disponibilizadas para resolver o quebra-cabeças, todas terão que iniciar exatamente do mesmo ponto e passar por todas as fases do problema.

Uma característica interessante desse quebra-cabeça, é que a sua complexidade é linear, ou seja, o tempo de execução (medido como $O(n)$) aumenta linearmente de acordo com o tamanho da entrada.

4.1.1 Introdução ao modelo

Esta abordagem foi uma das primeiras utilizadas para manter o sigilo de documentos eletrônicos a longo prazo.

4.1.2 Fase de construção

Ao construir um *Time Lock Puzzle*, o servidor quer cifrar uma mensagem M e mantê-la cifrada por um período de tempo T. Para isso, o servidor deve determinar quantas operações de quadrado o cliente terá que executar para este período de tempo T. Para tal, o servidor obtém do cliente a sua capacidade de operações de quadrado por segundo. Essa quantidade é o valor S. Uma vez conhecendo S, o servidor determina o valor de t, isto é, $t = TS$.

Feito isso, o servidor procede a construção do quebra-cabeças propriamente dito. O primeiro passo é gerar dois números primos grandes p e q, escolhidos aleatoriamente, e computar o valor de n, tal que $n = pq$. Agora o servidor deve escolher uma chave K grande e cifrar a mensagem utilizando um cifrador simétrico qualquer, por exemplo, AES (BAJAJ; GOKHALE, 2016).

$$C_M = AES(K, M) \tag{4.1}$$

Em seguida o servidor escolhe um valor a aleatoriamente, coprimo a n, tal que $1 < a < n$ e determina

$$b = a^{2^t} \mod n \tag{4.2}$$

A Equação 4.2 é o nosso quebra-cabeças. Resolver o quebra-cabeças significa determinar o valor de b. Como veremos, o servidor consegue resolver o quebra-cabeças de forma muito eficiente, diferentemente do cliente, que terá que fazer muitas operações de quadrado para resolvê-lo.

A resolução do quebra-cabeças (Equação 4.2) é muito custosa para valores de 2^t muito maiores que n. Entretanto, existe uma simplificação que é possível ser feita pelo servidor.

Usando o Teorema de Euler

$$a^{\phi(n)} \equiv 1 \pmod{n}$$

onde a é coprimo a n, consegue-se reduzir o expoente 2^t, ou seja, faz-se $2^t \pmod{\phi(n)}$. Para calcular o Totiente de Euler, o servidor faz:

$$\phi(n) = (p-1)(q-1) \tag{4.3}$$

O próximo passo deve ser a determinação do valor C_K, correspondente ao ciframento da chave K. Para isso, o servidor resolve o quebra-cabeças, adicionando o resultado a K, ou seja

$$C_K = K + b \mod n \tag{4.4}$$

Para chegar a este valor de maneira rápida, ele faz:

$$e = 2^t \mod \phi(n) \tag{4.5}$$

$$b = a^e \mod n \tag{4.6}$$

O resultado disto é um quebra-cabeça que contém os parâmetros (n, a, t, C_K, C_M), onde C_M é a cifra da mensagem, que poderá ser decifrada com a utilização da chave K, que só será descoberta depois de resolver o quebra-cabeça. Qualquer outra variável utilizada durante a geração do problema (como os primos p e q) é destruída.

4.1.3 Fase de solução

Para o cliente decifrar a mensagem M cifrada pelo servidor é necessário resolver o quebra-cabeça e encontrar a chave K que cifrou a mensagem M. Para resolver o quebra-cabeças, o cliente deve computar b através da Equação 4.7.

$$b = a^{2^t} \mod n \tag{4.7}$$

Com o valor de b em mãos, é necessário subtrair este valor do parâmetro C_K recebido. O resultado desta subtração é a chave K a ser utilizada para decifrar C_M e assim encontrar M.

O método mais eficiente para resolver a Equação 4.7 é através da computação de quadrados repetidos. Ver por exemplo (SCHNEIER,

2007).

A computação de quadrados repetidos tem complexidade proporcional do logaritmo na base dois do expoente, que neste caso é t, e este algoritmo não é paralelizável.

4.1.4 Exemplo

4.1.4.1 Construção do Quebra-cabeças

Para iniciar, devemos conhecer os valores T e S. Para este exemplo utilizaremos $S = 66.997$ e $T = 60$ segundos. Portanto, a quantidade de operações quadradas a serem executadas é:

$$t = 66.997 \times 60 = 4.019.820$$

.

Como números primos, vamos utilizar os valores $p = 7.901$ e $q = 7.639$. Sendo assim:

$$n = 7.901 \times 7.639 = 60.355.739$$

$$\phi(n) = 7.900 \times 7.638 = 60.340.200$$

Agora é necessário escolher um valor a, tal que $1 < a < n$. O valor escolhido é $a = 2$. Para construir o quebra-cabeça em si, iremos calcular, conforme a Equação 4.5:

$$e = 2^t = 2^{4.019.820} \pmod{60.340.200} = 17.693.776$$

$$
21\ quadrados \begin{cases}
2^{2^1} \mod 60.340.200 = 4 \\
2^{2^2} \mod 60.340.200 = 4^2 = 16 \\
2^{2^3} \mod 60.340.200 = 16^2 = 256 \\
2^{2^4} \mod 60.340.200 = 256^2 = 65.536 \\
2^{2^5} \mod 60.340.200 = 65.536^2 = 10.813.096 \\
2^{2^6} \mod 60.340.200 = 10.813.096^2 = 29.359.216 \\
2^{2^7} \mod 60.340.200 = 29.359.216^2 = 5.702.056 \\
\vdots \quad = \quad \vdots \\
2^{2^{19}} \mod 60.340.200 = 36.564.016^2 = 47.129.656 \\
2^{2^{20}} \mod 60.340.200 = 47.129.656^2 = 12.047.536 \\
2^{2^{21}} \mod 60.340.200 = 12.047.536^2 = 22.168.696
\end{cases}
$$

$$
\begin{aligned}
2^{4.019.820} \mod 60.340.200 = {} & 2^{2^{21}} \times 2^{2^{20}} \times 2^{2^{19}} \times 2^{2^{18}} \times 2^{2^{16}} \times 2^{2^{14}} \times \\
& 2^{2^{12}} \times 2^{2^{10}} \times 2^{2^9} \times 2^{2^6} \times 2^{2^5} \times 2^{2^3} \times 2^{2^2}
\end{aligned}
$$

Um vez que

$$
\begin{aligned}
4.019.820 = {} & 2^{21} + 2^{20} + 2^{19} + 2^{18} + 2^{16} + 2^{14} + 2^{12} + 2^{10} + 2^9 + 2^6 + \\
& 2^5 + 2^3 + 2^2 \\
= {} & 2097152 + 1048576 + 524288 + 262144 + 65536 + 16384 + \\
& 4096 + 1024 + 512 + 64 + 32 + 8 + 4
\end{aligned}
$$

O mesmo algoritmo é utilizado para calcular b, conforme a Equação 4.6:

$$b = a^e = 2^{17.693.776} \pmod{60.355.739} = 32.594.354$$

Para fazer essa exponenciação teremos mais 24 quadrados, totalizando 45 quadrados.

O próximo passo é computar C_K, utilizando a Equação 4.4. Para isto, utilizaremos o valor de chave $K = 171.721$:

$$C_K = 171.721 + 32.594.354 = 32.766.075$$

4.1.4.2 Resolução do Quebra-cabeças

Para solucionar o quebra-cabeça, o cliente recebe do servidor os valores ($n = 60.355.739, a = 2, t = 4.019.820$ e $C_K = 32.766.075$) e computa a Equação 4.7:

$$
\left.
\begin{array}{rl}
 & 2^{2^1} \mod 60.355.739 = 4 \\
 & 2^{2^2} \mod 60.355.739 = 4^2 = 16 \\
 & 2^{2^3} \mod 60.355.739 = 16^2 = 256 \\
 & 2^{2^4} \mod 60.355.739 = 256^2 = 65.536 \\
 & 2^{2^5} \mod 60.355.739 = 65.536^2 = 9.709.827 \\
 & 2^{2^6} \mod 60.355.739 = 9.709.827^2 = 6.169.853 \\
 & 2^{2^7} \mod 60.355.739 = 6.169.853^2 = 57.541.180 \\
 & \vdots \quad = \quad \vdots \\
 & 2^{2^{4.019.818}} \mod 60.355.739 = 58.152.121^2 = 11.308.679 \\
 & 2^{2^{4.019.819}} \mod 60.355.739 = 11.308.679^2 = 14.607.155 \\
 & 2^{2^{4.019.820}} \mod 60.355.739 = 14.607.155^2 = 32.594.354
\end{array}
\right\} 4.019.820 \textit{ quadrados}
$$

O único jeito de calcular esse valor de maneira mais rápida é descobrindo os valores p e q. Assim é possível solucionar o quebra-cabeça da mesma maneira eficiente com que o servidor fez, pois o cliente conseguirá encontrar o valor de $\phi(n)$, reduzir o expoente e acelerar o processo de resolução. Porém, para descobrir os valores p e q, é necessário fatorar n. Para casos onde o valor de n é pequeno esta fatoração pode ser fácil, porém, na medida que o tamanho dos números primos vai aumentando, esta fatoração torna-se muito difícil, fazendo com que computar $b = a^{2^t} \pmod{n}$ seja, de fato, o caminho mais fácil.

No protocolo descrito por Rivest, Shamir e Wagner (1996), o tamanho dos números é 2048 *bits* (mesmo valor utilizado na implementação deste trabalho), sendo assim impraticável encontrar uma solução através da fatoração de n.

4.2 SUBSET SUM

Este modelo de quebra-cabeça foi criado com o objetivo de atender à todas as propriedades fundamentais de um quebra-cabeça (TRITILANUNT et al., 2007). Foi primeiramente pensado para cenários de negação de serviço, que envolve trocas de mensagens frequentes entre cliente e servidor. Embora os quebra-cabeças feitos com o protocolo de *Subset Sum* contemplem, de fato, todas as propriedades desejáveis, este possui complexidade maior que linear. Isto quer dizer que o tempo de execução dele é definido por uma expressão polinomial ($T(n) = O(n^k)$), onde $K > 1$, fazendo com que o tempo varie de forma mais abrupta quando n é incrementado.

4.2.1 Introdução ao modelo

Esta abordagem é baseada no problema chamado "Problema da Mochila", como visto no Capítulo 2. Nesse esquema, o cliente recebe um problema semelhante ao da mochila para resolver. A ideia é selecionar um subconjunto de um conjunto de números fornecidos, tal que a soma dos números do subcojunto seja máxima e não ultrapasse um limiar estipulado pelo servidor.

Para que o método seja melhor compreendido daqui para frente, é necessário detalhar todas as variáveis que serão utilizadas neste protocolo:

- ID_I: Identificação do cliente;
- ID_R: Identificação do servidor;
- N_I: *Nonce* (número aleatório utilizado uma única vez) do cliente;
- N_R: *Nonce* (número aleatório utilizado uma única vez) do servidor;
- S: Parâmetro único e secreto escolhido para estabelecimento de comunicação;
- K: Dificuldade do problema;
- W: Limiar do quebra-cabeça;
- w_i: Peso do item i;

- n: Quantidade de itens do problema;
- $H()$: Operação *hash*;
- *Least Significant Bits* - $LSB(\alpha, K)$: Operação que obtém os K bits menos significativos de α;
- $C = LSB(\alpha, K)$

4.2.2 Fase de construção

Este quebra-cabeça é normalmente utilizado para evitar ataques de negação de serviço em um servidor. Um servidor só vai dar acesso a um cliente se este resolver um quebra-cabeça que será fornecido pelo servidor.

Quando um cliente quiser ganhar acesso a um determinado serviço de um servidor, este deve solicitar um quebra-cabeça, enviando junto à solicitação suas credenciais de identificação (ID_I, N_I). O servidor então vai armazenar as credenciais e escolher o parâmetro S de maneira aleatória. Este deve ser único, uma vez que irá identificar unicamente a comunicação entre cliente e servidor.

O próximo passo a ser executado pelo servidor é a escolha da dificuldade do problema. Quanto maior a dificuldade, mais tempo o cliente irá despender para solucionar o quebra-cabeça. Depois de escolher os valores de S e K o servidor vai fazer uma operação *hash* com os valores das cinco credenciais fornecidas como entrada $(ID_I, N_I, ID_R, N_R, S)$. O próximo passo é realizar a operação LSB para obter os K *bits* menos significantes do *hash* produzido. Esta operação dará origem à variável C:

$$C = LSB(H(ID_I, N_I, ID_R, N_R, S), K)_2{}^1 \tag{4.8}$$

Depois disto é necessário gerar o limiar do quebra-cabeça e conjunto de itens. Para gerar os itens, primeiro escolhe-se o item w_1 aleatoriamente. Depois disso, os próximos itens são criados fazendo *hash* do item anterior. Já para gerar o limiar W devemos considerar os bits de C, que serão representados por variáveis que vão de C_i até C_k. Cada um destes bits será multiplicado pelo item w_i correspondente, conforme

[1] O número 2 na operação representa a base, o que significa que a operação LSB pega os últimos bits, ao invés de os últimos bytes.

a Equação 4.9.

$$W = \sum_{i=1}^{k} C_i.w_i \tag{4.9}$$

O servidor irá então montar e enviar o quebra-cabeça, contendo os valores (w_1 (o primeiro item escolhido aleatoriamente), W (o peso máximo desejável) e K (dificuldade)), juntamente às credenciais do cliente e do servidor, para que o cliente possa verificar a autenticidade da comunicação.

4.2.3 Fase de solução

O primeiro passo, antes de tentar solucionar o problema proposto, é verificar as credenciais (ID_I, N_I, ID_R, N_R) enviadas pelo servidor junto com o quebra-cabeça, para checar se conferem com as credenciais enviadas anteriormente pelo cliente ao estabelecer conexão. O segundo passo a ser executado pelo cliente é a geração do conjunto de itens que será utilizado para a solução do problema. Para isto, o cliente obtém o parâmetro w_1, que representa o primeiro item da série de itens. Obtém também o valor K, que determina a quantidade de itens a serem gerados, e a partir daí pode obter todo o conjunto de itens subsequentes da mesma maneira que o servidor obteve: através de *hash* iterativo. Com o conjunto de itens na mão, o cliente pode iniciar o processo de resolução do quebra-cabeça.

O terceiro passo exige que o cliente construa um conjunto de bases B com os itens gerados, tal que:

$$b_1 = (1, 0, 0, ..., 0, w_1)$$
$$b_2 = (0, 1, 0, ..., 0, w_2)$$
$$b_3 = (0, 0, 1, ..., 0, w_3)$$
$$...$$
$$b_k = (0, 0, 0, ..., 1, w_k)$$
$$b_{k+1} = (0, 0, 0, ..., 0, -W)$$

É sobre este reticulado que será rodado o algoritmo LLL descrito

no Capítulo 2. O objetivo do algoritmo *LLL* é fazer uma redução de bases de reticulado, buscando encontrar a melhor base possível (HOFFSTEIN; PIPHER; SILVERMAN, 2008). Segundo Tritilanunt et al. (2007) o algoritmo em questão garante o retorno de um conjunto de vetores, onde o menor deve ser a solução para o quebra-cabeça. Para checar esta solução, o cliente deve multiplicar cada valor do vetor pelo item respectivo na lista de itens gerados, semelhante ao somatório feito pelo servidor em fase de construção.

Este vetor pequeno que representa o resultado será chamado de *C'*. Para averiguar se este é o correto o cliente irá checar se:

$$W \stackrel{?}{=} \sum_{i=1}^{k} C'_i.w_i \tag{4.10}$$

Onde C'_i corresponde a cada um dos valores do menor vetor encontrado. O valor encontrado no somatório não deve ultrapassar o limiar W. Se o resultado satisfizer esta condição, o cliente envia de volta ao servidor uma série de valores: as credenciais de identificação (ID_I, N_I, ID_R, N_R), o quebra-cabeça recebido anteriormente (w_i, W, K) e o vetor *C'* encontrado.

4.2.4 Fase de verificação

A fase de verificação é bem simples e barata para o servidor, eliminando assim as chances de sofrer um ataque de inundação nas soluções do quebra-cabeça, para cenários de negação de serviço. Segundo Tritilanunt et al. (2007) existem duas maneiras de verificar o resultado *C'* enviado pelo cliente, após verificar as credenciais recebidas:

- Evitando uso de *CPU*: O servidor mantém os valores corretos de C e W armazenados na memória juntamente à identidade do cliente, sendo necessário apenas uma consulta à tabela para conferir o resultado recebido com o armazenado previamente;

- Evitando uso de memória: Neste caso nenhuma informação é armazenada na memória do servidor até que o problema seja solucionado e enviado pelo cliente. Quando o resultado é recebido, o servidor gera novamente C e W, e compara com o resultado do cliente, o que é considerado uma operação barata e bem rápida.

4.2.5 Exemplo

4.2.5.1 Construção do Quebra-cabeças

O primeiro passo para a solução é gerar o conjunto de itens do problema. Para este exemplo, usaremos os valores $w = \{2, 4, 8, 9, 12\}$. O segundo passo é gerar o *hash* das credenciais de identificação, buscando os K *bits* menos significativos. Para um valor $K = 5$ vamos supor que os cinco *bits* menos significativos sejam 10010. Então $C = \{10010\}$. O servidor gera o limiar W através da Equação 4.10:

$$W = \sum_{i=1}^{5} C_i.w_i$$

$$W = (1 \times 2) + (0 \times 4) + (0 \times 8) + (1 \times 9) + (0 \times 12)$$

$$W = 2 + 0 + 0 + 9 + 12 = 11$$

O servidor envia os valores w_1, W e K para o cliente.

4.2.5.2 Resolução do Quebra-cabeças

O cliente irá gerar o conjunto de itens fazendo *hash* iterativo do primeiro item K vezes, assim como o servidor. Dado um conjunto de itens gerados $w = \{2, 4, 8, 9, 12\}$ e um limiar $W = 11$ estabelecido pelo servidor, o cliente deve contruir a matriz M abaixo:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 0 & 4 \\ 0 & 0 & 1 & 0 & 0 & 8 \\ 0 & 0 & 0 & 1 & 0 & 9 \\ 0 & 0 & 0 & 0 & 1 & 12 \\ 0 & 0 & 0 & 0 & 0 & -11 \end{pmatrix}$$

Agora é preciso fazer redução de base da mesma com o algoritmo LLL. Segundo Radziszowski e Kreher (1988), o menor vetor não nulo, encontrado pelo algoritmo LLL, deve ser constituído apenas de 0 e 1. Ao multiplicar o vetor pela matriz gerada, consideramos que os itens multiplicados por 0 não fazem parte da solução, enquanto os multiplicados por 1 fazem (seguindo assim a mesma lógica da multiplicação por *bits* feita na fase de construção).

O vetor retornado para este quebra-cabeça deve ser $v = \{1, 0, 0, 1, 0, 0\}$, pois, segundo Hoffstein, Pipher e Silverman (2008), a solução W deve ser igual à multiplicação de cada elemento do vetor por cada elemento da última coluna da matriz M, tal que:

$$W = M_1.v_1 + M_2.v_2 + M_3.v_3 + M_4.v_4 + M_5.v_5 + M_6.v_6$$

O que, neste caso, resulta em:

$$11 = (2 \times 1) + (4 \times 0) + (8 \times 0) + (9 \times 1) + (12 \times 0) + (-11 \times 0)$$

$$11 = 2 + 0 + 0 + 9 + 0 + 0$$

Confirmando assim que os pesos w_1 (2) e w_4 (9) são a resposta para o problema da mochila.

4.3 MODULAR SQUARE ROOTS

Esta técnica trouxe mais uma característica ao cenário de quebra-cabeças criptográficos. O que traz de novidade é a opção de uma abordagem não interativa (sem troca de dados sensíveis entre cliente e servidor) e ao mesmo tempo não paralelizável e com complexidade polinomial, assim como a abordagem anterior. Este é o primeiro dos protocolos expostos até então a contemplar estas três características simultaneamente (JERSCHOW; MAUVE, 2011).

Quando um problema é interativo (situação que ocorre em quase todas as abordagens conhecidas até hoje), o servidor envia o quebra-cabeça para o cliente junto com os parâmetros necessários para a solução do mesmo. Porém, pode haver um atacante no meio desta comunicação que consiga alterar os parâmetros, fazendo com que o cliente não consiga solucionar o problema proposto, segundo Jerschow e Mauve (2011).

Para isto, é proposto uma técnica que funciona tanto interativa quanto não interativamente, ou seja: é possível que o próprio cliente construa o problema matemático a ser resolvido, sem precisar receber

nada do servidor, evitando assim que haja um atacante alterando os parâmetros no meio da comunicação.

4.3.1 Introdução ao modelo

Para entender o funcionamento deste modelo, é necessário fazer uma revisão de como extrair raízes quadradas em módulo primo, conforme visto na Seção de Fundamentação Teórica. Este quebra-cabeça atende à propriedade de "não paralelização"utilizando o mesmo método dos outros dois previamente apresentados, pois é necessário o resultado do passo anterior para poder prosseguir para o próximo. Neste contexto, o servidor irá solicitar que o cliente realize uma repetição de operações de quadrados modulares de maneira sequencial, dentro de um corpo finito (JERSCHOW; MAUVE, 2011), como veremos a seguir.

4.3.2 Fase de construção

A abordagem explicada a seguir trata da técnica não interativa, onde o servidor não enviará o problema para o cliente. Ao invés disso, o próprio cliente irá construir o quebra-cabeça e solucioná-lo. Uma explicação mais detalhada é dada a seguir.

As duas partes (cliente e servidor) devem compartilhar previamente uma lista de números primos $p \equiv 1 \pmod 8$ com diferentes tamanhos em *bits*. Quando um cliente solicitar acesso a algum serviço de um servidor, este deve escolher um número primo da lista, de tamanho n *bits*, e aplicar *hash* iterativo sobre a mensagem de requisição. O processo de fazer *hash* iterativo ocorrerá c vezes, onde $c = \lceil \frac{n}{k} \rceil$. O valor k corresponde ao tamanho do *hash* que será gerado. O resultado desta divisão é arredondado para o número inteiro imediatamente superior. O resultado deste *hash* iterativo será d, que são os primeiros *n-1 bits* encontrados como saída das operações realizadas, como exposto a seguir:

$$d = First_{n-1}(H(m)||H(H(m))||...||H^c(m)) \tag{4.11}$$

Onde || representa a concatenação dos *hashs*. Agora é necessário verificar se d é um resíduo quadrático. Para isto, vai utilizar o símbolo de Legendre $(\frac{a}{p})$, onde p é o número primo escolhido inicialmente e $a = d$. Caso seja um resíduo não quadrático (resultado = -1), o cliente

deve decrementar o valor de a até encontrar um resíduo quadrático (resultado = 1). Este resíduo a é adicionado ao quebra-cabeça.

4.3.3 Fase de solução

Os valores recebidos na fase de solução são d e r. Para utilizar a nomenclatura do artigo original, a variável d é chamada de a em fase de solução. Para solucionar o quebra-cabeça, é necessário encontrar o valor de r, que é a raiz quadrada de a em módulo p, ou seja, computar:

$$\sqrt{a} \equiv r \pmod{p} \tag{4.12}$$

Para isto é utilizado o algoritmo *Cipolla-Lehmer* (LEHMER, 1969), que tem como entrada os valores d e p, e como saída as duas raízes quadradas de d em módulo p. Segundo Jerschow e Mauve (2011), o algoritmo *Cipolla-Lehmer* funciona da maneira explicada a seguir:

Primeiro o símbolo de Legendre é computado. Se o resultado for -1, o algoritmo conclui que a é um resíduo não quadrático módulo p e termina sua execução. Caso o resultado seja 1, o algoritmo prossegue escolhendo um número inteiro b aleatoriamente dentro do conjunto $\mathbb{Z}_p^*$, tal que o resultado de $b^2 - 4a$ seja um resíduo não quadrático módulo p, ou seja:

$$(\frac{b^2 - 4a}{p}) = -1 \tag{4.13}$$

Tendo os valores de a e b, cria o polinômio $f(x) = x^2 - bx + a$ em $\mathbb{Z}_p^*[x]$ e computa:

$$r = x^{\frac{(p+1)}{2}} \pmod{f} \tag{4.14}$$

O valor r encontrado deverá ser um inteiro. Por fim, o algoritmo retorna r e $-r$. Os autores da abordagem afirmam que, para o protocolo em questão, a raiz $-r$ não é importante. O cliente então envia ao servidor a raiz quadrada r extraída, juntamente à posição do número primo escolhido na lista compartilhada e a mensagem de requisição inicial. Desta maneira, não é preciso enviar explicitamente o número primo na comunicação.

4.3.4 Fase de verificação

Ao receber a requisição de acesso do cliente, bem como o quebra-cabeça resolvido (R), o servidor irá computar o valor d, da mesma maneira que o cliente computou, através da função de *hash* aplicada sobre a mensagem de requisição. Não é preciso utilizar o símbolo de Legendre para achar o resíduo quadrático a através do valor d encontrado, pois a probabilidade de $a = d$ é de 50%. Por consequência, a probabilidade de que $a = d - 1$ é de 25% (pois, caso $a \neq d$, é provável que o algoritmo tenha decrementado o valor encontrado em 1, como vimos anteriormente). Para verificar se a solução encontrada está certa, o servidor utiliza o resultado r recebido e computa:

$$d - (r^2 \pmod{p}) < \delta \tag{4.15}$$

Onde δ é uma constante pequena (como, por exemplo, 20). Caso seja, de fato, menor que esta constante, a solução é considerada correta. Embora o resultado recebido esteja certo, o servidor não necessariamente vai aceitar a requisição de acesso do cliente, pois primeiro irá avaliar a dificuldade (tamanho) do número primo p escolhido. Se o servidor considerar o número primo pequeno, ele pode negar o pedido de acesso, e o cliente pode escolher um novo número primo p maior da lista para tentar novamente.

4.3.5 Exemplo

4.3.5.1 Construção do Quebra-cabeças

O primeiro passo para construir o quebra-cabeça é escolher um número primo $p \equiv 1 \pmod 8$. O número escolhido foi 17, pois $17 \equiv 1 \pmod 8$. O segundo passo é fazer o *hash* da mensagem e pegar os primeiros $n - 1$ *bits* pra gerar o valor d. Para este exemplo, utilizaremos o valor $d = 9$. Agora precisamos verificar se este é um resíduo quadrático em módulo 17:

$$3^2 \equiv 9 \pmod{17}$$

Agora, que sabemos que 9 é um resíduo quadrático, já é possível iniciar a resolução do quebra-cabeça.

4.3.5.2 Resolução do Quebra-cabeças

Ao ter o valor de a e o valor de p, gerados na fase de construção, é necessário utilizar o algoritmo *Cipolla-Lehmer*: O primeiro passo do algoritmo é verificar se a é um resíduo quadrático:

$$3^2 \equiv 9 \pmod{17}$$

Em posse desta informação, devemos escolher o número inteiro b aleatório, desde que $b^2 - 4a$ seja um resíduo não quadrático. O número escolhido foi $b = 3$, pois, utilizando a Equação 4.13, verificamos que:

$$3^2 - 4 \times 9 = -27$$

$$-27 \equiv 7 \pmod{17}$$

Chegamos à conclusão de que 7 é um resíduo não quadrático em módulo 17, pois não há nenhum elemento em $\mathbb{Z}_{17}^*$ que seja sua raiz quadrada. Agora é necessário criar o polinômio $f(x) = x^2 - bx + a \in \mathbb{Z}_{17}^*[x]$ (anel dos números inteiros de 1 a 16):

$$f(x) = x^2 - 3x + 9$$

dado que $-3 \equiv 14 \pmod{17}$, o polinômio criado é:

$$f(x) = x^2 + 14x + 9$$

O próximo passo é computar o valor de r através de divisão polinomial expressa na Equação 4.14. O resto desta divisão será o valor r, enviado como solução ao servidor. Para fazer esta divisão, todo o valor de x deve ser um inteiro entre 1 e 16 (por ser uma divisão em módulo 17). Quando os valores ultrapassarem estes limites, devem ser colocados em módulo p:

$$
\begin{aligned}
r = x^{\frac{p+1}{2}} &= x^9 \pmod{f(x)} \\
&= f(x)(x^7) + 3x^8 + 8x^7 \pmod{f(x)} \\
&= f(x)(x^7 + 3x^6) + 7x^6 \pmod{f(x)} \\
&= f(x)(x^7 + 3x^6 + 7x^4) + 4x^5 + 5x^4 \pmod{f(x)} \\
&= f(x)(x^7 + 3x^6 + 7x^4 + 4x^3) + 15x^3 \pmod{f(x)} \\
&= f(x)(x^7 + 3x^6 + 7x^4 + 4x^3 + 15x) + 11x^2 + x \pmod{f(x)} \\
&= f(x)(x^7 + 3x^6 + 7x^4 + 4x^3 + 15x + 11) + 3 \pmod{f(x)} \\
r &\equiv 3 \pmod{f(x)}
\end{aligned}
$$

Para computar as equações acima, devemos pensar que, em cada linha, assumimos que $x^9 \pmod{f(x)}$ é equivalente à multiplicar $f(x)$ (ou seja, $x^2+14x+9$) por um valor $h(x)$ (na segunda linha, $h(x) = x^7$) e somar ao seu resto $r(x)$ (na segunda linha, $r(x) = 3x^8 + 8x^7$) em módulo $f(x)$, seguindo o padrão:

$$
\begin{aligned}
r &= x^9 \pmod{f(x)} \\
x^9 &= h(x)f(x) + r(x)
\end{aligned}
$$

Através da divisão de polinômios descobrimos que $r = 3$. Portanto, $-r = -3 \equiv 14 \pmod{17}$. Neste ponto da abordagem o cliente deve enviar somente o valor $r = 3$ para o servidor conferir. O servidor, através da mesma função *hash* utilizada pelo cliente, chega ao valor $d = 9$. Para conferir a validade do resultado recebido, o servidor deve computar $d - (r^2 \mod p) < \delta$ para o valor de r recebido. Assim como no artigo original, definimos $\delta = 20$:

$$
\begin{aligned}
9 - (3^2 \mod p) &< 20 \\
9 - (9 \mod p) &< 20 \\
9 - 9 &< 20 \\
0 &< 20
\end{aligned}
$$

Assim o servidor pode confirmar a validade do valor $r = 3$.

Para entendermos a comparação com uma constante na fase de verificação, ao invés de fazermos uma igualdade, podemos pensar que o valor d (primeiros $n-1$ *bits* do *hash* gerado) é 10. Em fase de cons-

trução o cliente vai verificar que este valor não é um resíduo quadrático em módulo 17, e vai decrementar esse valor por 1, para então encontrar o valor $a = 9$. No entanto, o servidor não faz essa verificação se d é, ou não, um resíduo quadrático. Por este motivo, o valor d encontrado pelo servidor não será decrementado, e o resultado da equação $d - (r^2 \mod p) = \delta$ não será verdadeiro.

Por este motivo é necessário definir uma constante pequena para fazer comparação. O número 20 escolhido é uma boa constante, pois dificilmente será necessário decrementar um valor 20 vezes para encontrar um resíduo quadrático. No caso de $d = 10$, o servidor irá computar:

$$10 - (3^2 \mod p) < 20$$

$$10 - (9 \mod p) < 20$$

$$10 - 9 < 20$$

$$1 < 20$$

Assim o servidor pode confirmar a validade do valor $r = 3$.

5 TECNOLOGIAS UTILIZADAS PARA IMPLEMENTAÇÃO DOS QUEBRA-CABEÇAS

Este capítulo descreve as tecnologias utilizadas para a implementação das três abordagens de quebra-cabeças criptográficos vistas no capítulo 4. A Seção 5.1 apresenta a linguagem de programação utilizada neste trabalho para implementar os quebra-cabeças. Esta linguaguem é de fácil aprendizado e possui bibliotecas nativas que auxiliaram a implementação das partes baseadas em matemática mais complexa. Para os algoritmos matemáticos mais complexos, como LLL visto anteriormente, foi necessário importar uma biblioteca externa, que será descrita na Seção 5.2

5.1 LINGUAGEM DE PROGRAMAÇÃO

A linguagem de programação escolhida para implementar as abordagens de quebra-cabeça criptográfico detalhadas no Capítulo 4 foi *Python* 3.5 (PYTHON, 2001). Esta linguagem dispõe de bibliotecas muito boas que contém métodos importantíssimos para o contexto tratado, agilizando a programação em várias etapas, como por exemplo na utilização do algoritmo *LLL* (LENSTRA; LENSTRA; LOVÁSZ, 1982) que trata de redução de bases de reticulado.

Além disso, esta linguagem tem crescido muito nos últimos anos, por se tratar de uma linguagem funcional e de fácil sintaxe, trabalhando de modo bastante intuitivo com programadores que já têm experiência prévia em outras linguagens. É uma linguagem muito simples para produzir e rodar testes, sendo também de fácil, simples e rápida instalação em qualquer sistema operacional. No caso desta pesquisa, o sistema utilizado é o *Windows* 8 (MICROSOFT, 1975) em uma máquina 64 *bits*. O código foi todo produzido na ferramenta de edição *Atom* (ATOM, 2015), versão 1.14.1, e os testes foram rodados diretamente via terminal (*cmd*).

Para a abordagem de *Modular Square Roots*, no entanto, o ambiente de desenvolvimento não foi o *Atom*, e sim o *Sage* (SAGE, 2005), um software de matemática que possui recursos para operações com álgebra, combinatória, análise numérica, teoria dos números e cálculo. O software suporta a linguagem Python, na qual a abordagem foi de-

senvolvida. A preferência por este ambiente está no fato de que esta abordagem opera com polinômios sobre corpos finitos, e o Sage possui bibliotecas ótimas e de fácil utilização para cálculos com polinômios em anel.

Por se tratar de uma aplicação onde, até o presente momento, não há interface definida de comunicação com o usuário (afinal os testes foram feitos de maneira unitária), não houve nenhuma linguagem relacionada à visualização de dados, tampouco servidores envolvidos para publicação do programa em alguma plataforma.

5.2 BIBLIOTECA UTILIZADA

Para a implementação de todas as três abordagens de quebra-cabeça criptográfico foi utilizada somente uma biblioteca do *Python*: *L3.py*, que forneceu métodos para a classe *SubsetSum.py*. De acordo com o que vimos no Capítulo 4, utilizamos os métodos desta biblioteca para realizar três passos do algoritmo: criar o conjunto de bases B que forma o reticulado, reduzí-lo com o algoritmo LLL, retornando assim um conjunto de vetores pequenos, e, por fim, encontrar o menor vetor possível no conjunto de vetores retornados.

Estes foram os únicos passos onde se usou uma biblioteca importada dentro do programa. A classe *TimeLock.py* não exigiu o uso de nenhuma biblioteca externa, visto que os conceitos matemáticos que ela utiliza são nativos na própria linguagem. A classe *ModularSquareRoots.sagews* utilizou conceitos nativos do sage.

Além da biblioteca importada no programa, importamos alguns módulos já existentes dentro da linguagem, para poder utilizar criptografia nas partes necessárias e para algumas transformações de dados que se fizeram necessárias.

6 IMPLEMENTAÇÃO DOS QUEBRA-CABEÇAS CRIPTOGRÁFICOS NÃO PARALELIZÁVEIS

O objetivo inicial com a criação desta aplicação é testar as abordagens detalhadas no Capítulo 4, compreendendo os conceitos matemáticos envolvidos, bem como testar cada uma delas. Além disso, pretende-se que esta possa estar disponível para qualquer um que queira entender, utilizar ou testar os conceitos de quebra-cabeças criptográfico que possua a propriedade de não paralelização. Vale salientar que a propriedade de não paralelização é muito importante e não é utilizada somente em casos onde envolve o fator tempo.

No contexto estudado neste trabalho, por exemplo, apesar de as implementações de *Subset Sum* e *Modular Square Roots* possuírem esta propriedade, elas foram voltadas para o cenário de ataques de negação de serviço. Sendo assim, a implementação da primeira abordagem vista (*Time Lock Puzzle*) é a única onde é possível fornecer o tempo exato de solução do quebra-cabeça. As outras duas, embora não paralelizáveis, não permitem que seja selecionado um tempo específico para resolução. Podemos controlar apenas a dificuldade para solução de ambas, fazendo com que os testes mais difíceis demorem mais para serem resolvidos.

Tendo isto em mente, foi preciso planejar cuidadosamente como esta aplicação seria estruturada. Sabia-se que era necessário receber os parâmetros principais do usuário, como tamanho de chave, dificuldade do problema e mensagem a ser cifrada, nos casos em que esta se faz necessária.

Para atender a todos os objetivos listados acima e possibilitar que o programa seja utilizado em qualquer contexto, ampliando assim a sua abrangência, foi criada uma estrutura concisa e bastante objetiva, de maneira que, ao olhar o código, o usuário consiga saber de qual abordagem se trata e qual o objetivo de cada método apresentado.

A aplicação foi desenhada para dar a maior sensação de simplicidade possível, abrindo-se assim as portas para dar continuidade a esta pesquisa no futuro, estendendo e aprimorando o código criado.

6.1 ORGANIZAÇÃO DO CÓDIGO

De acordo com o que foi descrito no item acima, a estrutura e a organização do código foram pensadas com cuidado para que tudo ficasse, ao mesmo tempo, simples e intuitivo. Para evitar que hajam dúvidas quanto à que abordagem pertence cada um dos algoritmos criados, a aplicação foi organizada em dois pacotes diferentes, que levam os nomes dos protocolos *time_lock* e *subset_sum*. A última das três abordagens foi desenvolvida separadamente, em um único arquivo, no ambiente *sage*. O arquivo levou o nome *ModularSquareRoots.sagews*.

Dentro de cada um dos pacotes, no que se refere às duas primeiras abordagens estudadas, encontramos, no mínimo, três arquivos, que correspondem a: arquivo obrigatório de inicialização do *Python* (gerado automaticamente para cada pacote), classe que contém os métodos necessários para construção, solução e verificação do quebra-cabeça e um último arquivo de testes unitários, que contém, para cada classe, uma série de testes que medem a performance dos algoritmos. O pacote *subset_sum* possui um arquivo extra, *L3.py*, que diz respeito à biblioteca externa utilizada. É possível visualizar a estrutura na Figura 3.

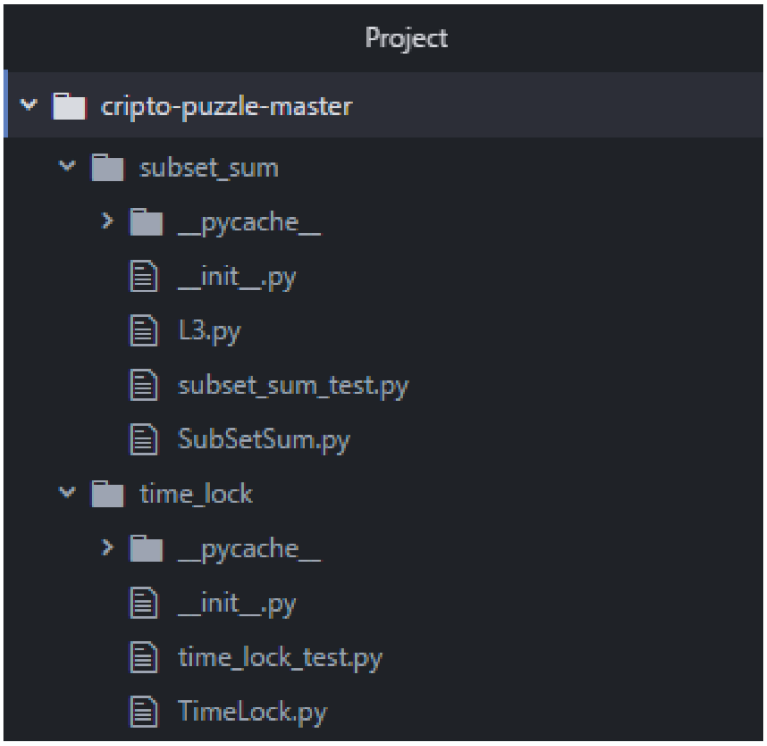

Figura 3 – Organização do código para Time Lock e Subset Sum

É importante esclarecer que, por motivos de padrão internacional, o código e os testes foram escritos em inglês, facilitando o entendimento para qualquer tipo de usuário, de qualquer lugar, visto que trata-se de uma língua universal. Além disso, isto facilita para que futuros projetos possam ser desenvolvidos para estender o presente trabalho, pois o campo de segurança em computação é muito amplo e ainda há muito trabalho que pode ser feito em cima deste código, como veremos posteriormente na Seção 8.1 (ver página 84) onde são descritos os trabalhos futuros.

6.2 DESCRIÇÃO DA IMPLEMENTAÇÃO

Esta seção será dividida da seguinte maneira: para cada uma das três classes, divididas entre os três pacotes, será descrito, no mínimo, o método de inicialização (que recebe parâmetros e carrega algumas variáveis indispensáveis para o algoritmo), o método que cria o quebra-cabeça, e o que resolve o mesmo. Os testes produzidos para cada classe deram origem aos resultados experimentais, que serão expressos no Capítulo 7.

6.2.1 Time Lock

A primeira classe implementada foi a do algoritmo *TimeLock*. O primeiro método que deve ser chamado, antes mesmo de criar o quebra-cabeça, é o método *get_ s*, que vai calcular quantas operações de quadrado modular a máquina consegue fazer por segundo, pois este valor é necessário na fase de construção. A Figura 4 apresenta este método.

```
def get_s(self):
  totalsquares = 0
  rsa_ = self.get_rsa()
  for num in range(0,25):
    numbern = rsa_.private_numbers().public_numbers.n
    numbera = ramdom.SystemRamdom().randint(1, numbern)
    spersecond = self.compute_s(numbera, numbern)
    totalsquares = totalsquares + spersecond

  totalsquares = totalsquares/25
  return totalsquares
```

Figura 4 – Método *get_ s* para Time Lock

Ainda neste método criamos, a cada iteração, um número primo n de 2048 *bits* e um número a, tal que $1 < a < n$, e chamamos o método *compute_ s* para calcular quantas operações de quadrado modular é possível realizar em 1 segundo. Em nossos experimentos, foram feitas 25 iterações que, ao final, retornam uma média mais precisa da quantidade de operações por segundo. A quantidade de iterações foi pensada de maneira que não demorasse muito para realizar o cálculo, mas ao mesmo tempo tivessem iterações suficientes para uma média mais precisa do número de operações por segundo.

```
def compute_s(self, a, n):
  start_time = time.time()
  num_squares = 100000
  for num in range(0, num_squares):
    b = pow(a, 2, n)
  end_time = time.time()
  tot_time = end_time-start_time
  return num_squares/tot_time
```

Figura 5 – Método *compute_ s* para Time Lock

Para calcular quantas operações de quadrado modular é possível realizar em 1 segundo, o método *compute_ s* (a Figura 5 apresenta este método) resolve 1.000.000 operações de quadrado modular em um determinando tempo, retornando a média por segundo. A execução destes dois métodos dá origem ao parâmetro S da abordagem.

Agora é preciso inicializar algumas variáveis que serão utilizadas na construção do quebra-cabeça. O método *create_ variables*, mostrado na Figura 6, recebe como parâmetro a mensagem a ser cifrada, o tempo para solução em segundos e o valor obtido na saída do método *get_ s*.

```
def create_variables(self, message, seconds, totalsquares):
  rsa_ = self.get_rsa()
  self.n = rsa_.private_numbers().public_numbers.n

  self.M = bytes(message)
  self.T = int(seconds)
  self.S = int(totalsquares)

  p = rsa_.private_numbers()._p
  q = rsa_.private_numbers()._q

  self.phi = (p - 1) * (q - 1)
  self.t = self.T * self.S
  self.K = self.gen_random_n_digits(16)
```

Figura 6 – Método *create_ variables* para Time Lock

As variáveis são inicializadas com os respectivos valores e recebem a mesma nomenclatura presente no algoritmo original. É neste método também que os números primos de 2048 *bits* são criados, dando origem ao valor $\phi(n)$. O próximo passo é criar o quebra-cabeça, como exposto na Figura 7.

```
def create_puzzle(self):
  self.a = random.SystemRandom().randint(1, self.n)
  self.iv = os.urandom(16)
  K_bytes = self.to_bytes(self.K)
  enc = Cipher(
      algorithms.AES(k_bytes),
      modes.OFB(self.iv),
      backend = default_backend()
  ).encryptor()
  self.C_M = enc.update(self.M) + enc.finalize()

  e = pow(2, self.t, self.phi)
  b = pow(self.a, e, self.n)
  self.C_K = self.K + b
  return self.C_M, self.iv, self.C_K, self.a, self.t, self.n
```

Figura 7 – Método *create_puzzle* para Time Lock

No método *create_puzzle*, mostrado acima, criamos um objeto cifrador com o algoritmo *AES*, que irá utilizar a chave K inicializada previamente e um *iv* aleatório para cifrar a mensagem armazenada (o *iv* não faz parte da abordagem original, porém é obrigatória a utilização do mesmo ao usar *AES* no python), dando origem à C_M.

O AES (*Advanced Encryption Standard*) é um padrão de criptografia avançado selecionado pelo NIST (*National Institute of Standards and Technology*) (NIST, 1901) em 2001. Trata-se de um algoritmo criptográfico (algoritmo de Rijndael (DAEMEN; RIJMEN, 1999)) utilizado para fins de proteção e segurança de dados compartilhados e mantidos em meio eletrônico (BAJAJ; GOKHALE, 2016).

Agora é necessário cifrar também a chave privada K. Os passos para isto seguem à risca o que foi detalhado no Capítulo 4: Primeiro é gerada a variável a que, elevada à variável e, dá origem à variável b. O método *pow* do python faz esta exponenciação e já coloca o resultado em módulo (o módulo vai depender do número enviado como último parâmetro do método *pow*). Com isto criamos a variável C_K. O método vai retornar os parâmetros necessários para resolução do quebra-cabeça, conforme visto previamente no detalhamento do algo-

ritmo. Aqui há uma leve diferença do artigo original, pois reforçamos a segurança concatenando um *iv* à chave inicial, portanto este *iv* precisa ser enviado agora também.

Tendo o retorno do método *create_puzzle* é possível começar a solução do quebra-cabeça, através do método *solve*, que pode ser visto na Figura 8.

```
def solve(self, iv, message, C_K, a, t, n):
  b = self.compute_b(a, t, n)
  key = C_K - b
  dec = Cipher(
      algorithms.AES(self.to_bytes(key)),
      modes.OFB(iv),
      backend = default_backend()
  ).decryptor()
  return dec.update(message) + dec.finalize()

def compute_b(self, a, t, n):
  return pow(a, pow(2, t), n)
```

Figura 8 – Método *solve* para Time Lock

O método acima computa o valor b chamando um segundo método *compute_b*, também exposto na Figura 8. Sabendo o valor de b, é possível subtrair de C_K e encontrar a chave privada K. Com esta chave secreta e o *iv* podemos decifrar a mensagem e comparar com a original.

6.2.2 Subset Sum

Para construir o quebra-cabeça, neste caso, só é necessário enviar a sua dificuldade, denominada K. O construtor então inicializa esta variável e cria ID_r e N_r, que representam a identidade e o *nonce* (número único) do servidor. As credenciais de identificação, tanto do servidor quanto do cliente, devem ser criadas aleatoriamente.

O próximo passo é criar o quebra-cabeça, recebendo como parâmetro as credenciais do cliente, denominadas ID_i e N_i. O método de criação, cuja assinatura é *create_puzzle*, cria o valor S, que identifica unicamente a comunicação entre duas partes, de maneira aleatoria, e faz *hash* de todas as credenciais, obtendo assim a variável *hash*.

O valor C é então gerado, pegando os K *bits* menos significativos da variável *hash* obtida anteriormente. Agora, para gerar o peso máximo W desejado, é necessário gerar a série de itens que serão utilizados na fase de solução. Para isto, o método *get_w* é chamado, que irá retornar esta coleção. Com isto é possível gerar o quebra-cabeça, multiplicando cada valor de C pelo seu respectivo valor na lista w de pesos. É possível ver o somatório na Figura 9. A função *enumerate* do python serve para iterar sobre listas com índice numérico, que neste caso é a variável C.

```
def create_puzzle(self, IDi, Ni):
  self.S = self.cryptogen.randint(1000, 999999)

  hash = self.get_hash(IDi, Ni, self.Nr, self.IDr, self.S)
  hash = bin(int(hash, 16))[2:].zfill(8)

  self.C = hash[-self.K:]
  self.C = [int(i) for i in self.C]

  w = self.get_w(self.K)

  W = 0
  for idx, val in enumerate(self.C):
      W += self.C[idx] * w[idx]

  return w[0], W, self.K
```

Figura 9 – Método *create_puzzle* para Subset Sum

O método para solução do quebra-cabeça segue o mesmo padrão da abordagem anterior, ou seja, os parâmetros de entrada são exatamente os de saída do método de construção. O primeiro passo é criar todos os pesos da lista, aplicando função *hash* sobre o peso imediatamente anterior, k vezes, começando com o item w_0 recebido no retorno do método de construção. O próximo passo requer o uso da primeira biblioteca, para que possamos criar a matriz inicial do problema (função *create_matrix_from_knapsack_orig* no método *solve*), que é a entrada para o algoritmo LLL. Conforme visto no Capítulo 4, a entrada para esta primeira função da biblioteca deve ser a lista de pesos e o peso máximo suportado, ou seja: w e W, como podemos ver na Figura 10.

Com essa matriz em mãos, composta por bases, fazemos a redução dela com o algoritmo *LLL* (função *lll_reduction* no método *solve*) e, a partir dela, recebemos um conjunto de vetores. Este conjunto de vetores é submetido ao método *best_vect_knapsack* da biblioteca *L3.py*, que retorna o menor vetor do conjunto, considerado a resposta

```
def solve(self, wi, W, k):
  w = [wi]
  i = 0
  while i < k-1:
      w.append(int(self.get_hash(w[i]), 16))
      i += 1

  matrix = create_matrix_from_knapsack_orig(w, W)
  reduced = lll_reduction(matrix)
  solution = best_vect_knapsack(reduced)

  return solution
```

Figura 10 – Método *solve* para Subset Sum

para o problema. O caminho de verificação é o mesmo do algoritmo original, comparando a variável obtida no método *solve* com o valor de C armazenado previamente na memória.

6.2.3 Modular Square Roots

Nesta abordagem o cliente precisa escolher um número primo $p \equiv 1 \pmod 8$ para iniciar a construção do quebra-cabeça. Para isto, no método *create_puzzle* o cliente pode enviar o tamanho em *bits* que deseja que o número primo tenha, e o método *get_n_bit_prime* vai retornar um número primo congruente a 1 mod 8 do tamanho especificado. Com este número em mãos, o cliente segue fazendo *hash* da mensagem, que é um dos parâmetros de entrada para a construção do problema. Para calcular se o valor d encontrado é um resíduo quadrático, chama o método *calculate_legendre*. Este método é o único desta abordagem que foi reaproveitado, pois existem várias fontes para o mesmo. Se não for um resíduo quadrático, o valor será decrementado por 1 até encontrar um que seja. O método *create_puzzle* pode ser observado na Figura 11.

O próximo passo é chamar o método *solve*. Neste caso, a única coisa que ele faz é chamar o método *cipolla_lehmer*, que vai retornar o valor de r, dito como a solução do quebra-cabeça. O método *cipolla_lehmer* pode ser observado na Figura 12, junto ao método *find_b*, que será utilizado para encontrar o valor de b, tal que o resultado de $b^2 - 4a$ seja um resíduo não quadrático. Para saber se b é não quadrático, o método *find_b* também utiliza o método *calculate_legendre*. O método *find_b* então retorna um conjunto de possíveis valores b, no

```
def create_puzzle(message, nbits):
    p = get_n_bit_prime(nbits)
    n = number_of_bits(p)

    hash_final = get_d(message, n)
    hash_bits = bin(hash_final)
    first_bits = hash_bits[2:n+1]
    d = int(first_bits, 2)

    is_residue = calculate_legendre(d, p)
    while is_residue != 1:
        d = d-1
        is_residue = calculate_legendre(d,p)

    return d, p
```

Figura 11 – Método *create_puzzle* para Modular Square Roots

qual um é escolhido aleatoriamente pelo método *cipolla_lehmer*. Tendo os valores de a, b e p, cria o polinômio $x^2 - bx + a$ dentro de um corpo finito com elementos de 1 a 16 e opera sobre ele. Estes últimos passos foram feitos na sintaxe do sage.

```
def cipolla_lehmer(a, p):
    b = find_b(a, p)
    max_list = len(b)-1
    list_position = randint(0,max_list)
    b = b[list_position]

    F.<x> = GF(p)[]

    f = x**2 -b*x + a
    r = x**( (p+1)/2 ) % f

    return r

def find_b(a, p):
    set_of_b = []
    for b in range(1, p):
        possible_b = b**2 - 4*a
        nonresidue = calculate_legendre(possible_b, p)
        if nonresidue == -1:
            set_of_b.append(b)
    return set_of_b
```

Figura 12 – Método *cipolla_lehmer* para Modular Square Roots

Como entrada do método de verificação dos resultados, temos o resultado r, a mensagem inicial e o primo p. O método *check* vai pegar o tamanho em *bits* do parâmetro p para gerar o *hash* da mensagem recebida e pegar os $n-1$ primeiros *bits* da saída da função *hash*. Como

explicado no Capítulo 4, não é necessário calcular se é resíduo quadrático. A única verificação é feita escolhendo uma constante (neste caso escolhemos 20, para manter o padrão do algoritmo original) e verificando se esta é maior que o valor $d - (r^2 \pmod{p})$. Os passos para a verificação do quebra-cabeça podem ser observados na Figura 13.

```
def check(message, r, p):
    n = number_of_bits(p)

    hash_final = get_d(message, n)
    hash_bits = bin(hash_final)
    first_bits = hash_bits[2:n+1]
    d = int(first_bits, 2)

    delta = 20
    print d - (int(r)^2)%p < delta
```

Figura 13 – Método *check* para Modular Square Roots

Os métodos apresentados neste capítulo são os principais para o entendimento do contexto, porém existem métodos secundários que servem de apoio, seja para cálculo de *hash*, para transformação de objetos *string* em binários ou para fazer iterações sobre listas. No entanto, os conceitos tratados no Capítulo 4 foram todos abordados e serão testados no Capítulo 7.

Vale salientar que, caso haja a necessidade ou curiosidade por parte do leitor de verificar toda a implementação, o código em python está disponível na íntegra como primeiro anexo deste trabalho.

7 RESULTADOS EXPERIMENTAIS

7.1 INTRODUÇÃO

Depois de concluída a implementação dos algoritmos, foi necessário rodar os testes produzidos e observar se eles se comportavam de maneira adequada, utilizando a mesma máquina utilizada durante a fase de implementação. A mesma simulou o comportamento do servidor e do cliente. Através da execução dos testes também foi possível medir a performance de cada algoritmo, de acordo com os parâmetros de entrada, bem como tirar conclusões sobre a eficiência e aplicabilidade de cada um, tendo em vista os tempos de execução, tanto para construção como para solução. Para cada uma das abordagens foram executados mais de 20 testes, porém, para tornar este capítulo mais objetivo, vamos expor os resultados encontrados em apenas uma parte dos testes. Para cada abordagem, serão mostrados os valores de, pelo menos, dez testes realizados. O resto deles seguiu o mesmo padrão.

7.2 TIME LOCK

Ao rodarmos os testes para a abordagem de *Time Lock*, sabíamos que o tempo de solução do quebra-cabeça deveria ser muito próximo ao estipulado, podendo ter pequenas variações de tempo, conforme o artigo original aceita (RIVEST; SHAMIR; WAGNER, 1996).

Teste	**Tempos [Segundos]**		
	Fornecido	**Construção**	**Solução**
1	1	0,03	0,92
2	2	0,04	1,84
3	40	0,03	39,98
4	70	0,04	71,12
5	85	0,04	85,05
6	115	0,04	115,73
7	130	0,03	130,05
8	600	0,03	579,07
9	10.800	0,03	10.522,47
10	18.000	0,04	17.948,25

Tabela 1 – Testes para Time Lock

A Tabela 1 mostra, para cada teste executado, o tempo fornecido em fase de construção (que corresponde ao tempo em que o quebra-cabeça deve ser resolvido), o tempo que levou para ser cons-

truído e o tempo que levou para ser solucionado. Os valores da coluna "Solução" devem ser muito próximos aos da coluna "Fornecido".

Um *screenshot* de uma das rotinas de testes pode ser observado na Figura 14. Antes de mostrar os resultados, o programa nos informa a quantidade de operações de quadrado em módulo nossa máquina está processando no momento.

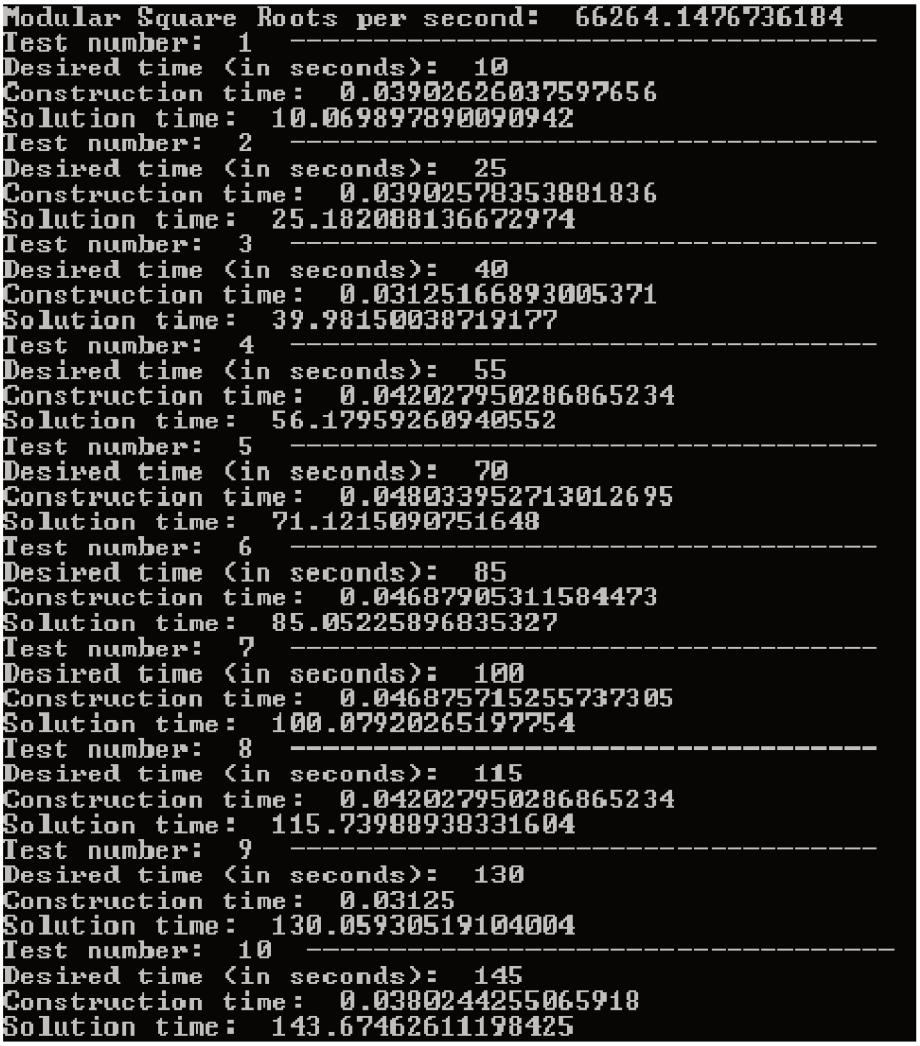

Figura 14 – Testes para Time Lock

Os resultados experimentais encontrados para esta abordagem foram muito satisfatórios, visto que os valores foram muito aproximados, com a diferença de tempo girando entre 10 e 40 milésimos de segundos nos casos onde o tempo era muito pequeno. Já nos testes em que a máquina ficou rodando por grandes quantias de tempo, a diferença foi um pouco maior. No caso em que o quebra-cabeça deveria ser desvendado em 3 horas, este foi solucionado 4 minutos antes, que

foi a maior diferença encontrada, como podemos observar na Tabela (1). Já no caso onde o tempo fornecido foi 5 horas (18000 segundos), o erro foi de apenas 52 segundos. Para aumentar a precisão dos resultados encontrados, cada um dos testes foi executado mais de uma vez, produzindo também uma média do tempo de solução para cada caso. No maior dos casos testados, para 5 horas, a média retornada para 10 execuções deste teste foi 17.739,08.

Ao analisar o tempo de geração do quebra-cabeça, os resultados também são ótimos, visto que seguem a regra fundamental que diz que o problema deve ser fácil de construir e difícil de resolver. Ao olharmos a coluna "Construção" observamos que nenhum quebra-cabeça demorou mais do que 0,04 milésimos de segundo para ser construído, não interessando se o parâmetro de entrada é 10 segundos, ou 18000 segundos.

7.3 SUBSET SUM

Nesta abordagem os resultados devem ser interpretados de maneira diferente, já que não há como prever o tempo exato em que serão solucionados. A característica forte deste protocolo é que o tempo para solucionar o quebra-cabeça deve aumentar na medida que a dificuldade fornecida nos testes aumenta. Iniciamos nossa rotina de teste com o valor de dificuldade 10 e aumentamos este até 65. Os valores de solução, expostos em segundos, são baixos, mas cresceram rapidamente. O resultados dos testes é mostrado na Tabela 2.

Teste	**Tempos [Segundos]**		
	Dificuldade	**Construção**	**Solução**
1	10	0,00	1,25
2	11	0,00	1,56
3	19	0,01	6,79
4	20	0,00	7,96
5	29	0,00	21,10
6	32	0,00	26,87
7	33	0,00	40,51
8	39	0,00	50,12
9	40	0,00	67,34
10	50	0,01	84,33
11	60	0,00	118,74

Tabela 2 – Testes para Subset Sum

Os resultados também foram satisfatórios, visto que o quebra-

cabeça obedece a proporção de aumento entre a dificuldade e o tempo. Já ao analisarmos a coluna "Construção", concluimos que a abordagem é ainda mais eficiente para o servidor do que a anterior, visto que a geração do problema é quase instantânea, fazendo com que o servidor não tenha que desprender muitos recursos nesta fase.

A quantidade de tempo é bem pequena, porém segue a mesma linha da versão original criada por Tritilanunt et al. (2007). Ao testarem a própria abordagem os autores concluíram que para qualquer dificuldade menor que 25 o tempo de solução do quebra-cabeça era de 0.0 segundos, ou seja: instantâneo. Para casos onde a dificuldade era maior que 25 e menor que 50 o tempo ainda era muito baixo (existindo casos de solução em 0.1 segundos, como é possível observar na Tabela 3). Os autores só conseguiram um certo nível de dificuldade a partir do valor 50. Ainda segundo os autores da abordagem, não é recomendado usar $K > 100$, pois o tamanho da matriz de reticulado construída é grande demais, esgotando os recursos de memória.

Itens	Tempo Médio de Execução [segundos]																	
	Conjunto Aleatório 1						Conjunto Aleatório 2						Conjunto Aleatório 3					
	Densidade						Densidade						Densidade					
	0,3	0,4	0,5	0,6	0,7	0,8	0,3	0,4	0,5	0,6	0,7	0,8	0,3	0,4	0,5	0,6	0,7	0,8
60	0,10	0,12	0,23	1,02	2,42	77,11	0,16	0,28	0,19	0,31	3,64	3,70	0,14	0,22	0,21	0,61	0,64	3,21
65	0,14	0,14	0,29	1,59	4,09	190,68	0,18	0,29	0,23	0,57	6,53	6,86	0,17	0,23	0,26	1,70	2,19	18,94
70	0,15	0,15	0,32	2,94	7,33	342,53	0,18	0,29	0,28	1,34	12,97	26,30	0,21	0,25	0,27	2,29	2,29	41,72
75	0,20	0,14	0,78	5,23	13,47	663,24	0,24	0,31	0,38	1,95	27,23	35,65	0,23	0,25	0,34	3,49	4,37	92,37
80	0,27	0,22	0,89	9,63	26,17	1745,97	0,25	0,33	0,52	2,75	58,70	87,12	0,26	0,29	0,45	5,66	8,82	226,76
85	0,37	0,25	1,24	17,38	49,22	4158,73	0,29	0,37	0,72	4,44	120,44	208,86	0,28	0,32	0,62	9,40	18,15	1315,29
90	0,50	0,29	1,63	31,44	96,39	9435,02	0,39	0,40	1,17	7,58	250,52	509,60	0,30	0,37	0,89	16,42	37,75	1344,35
95	0,59	0,34	2,34	55,68	173,30	21351,72	0,43	0,43	1,75	12,78	504,88	1158,45	0,36	0,43	1,28	28,14	79,36	3160,86
100	0,70	0,40	3,43	98,39	317,27	51124,86	0,46	0,47	2,87	21,45	1008,23	2737,79	0,41	0,50	2,03	46,63	168,72	7451,26

Tabela 3 – Resultado Experimental do Subset Sum (TRITILANUNT et al., 2007)

7.4 MODULAR SQUARE ROOTS

Já esta abordagem deve ser analisada de acordo com o número primo escolhido para geração do problema. Quanto maior o número, maior deve ser o tempo desprendido para a solução.

Assim como nas abordagens anteriores, podemos comprovar, através dos testes, que a aplicação funciona de acordo com o desejado. O tempo para solução do quebra-cabeça cresce de acordo com a dificuldade de encontrar a raíz quadrada de um elemento módulo um número primo. Isto acontece pois, quanto maior o número primo, maior a quantidade de elementos a serem testados dentro do corpo finito.

Teste	Tempos [Segundos]		
	Número primo	**Construção**	**Solução**
1	17	0,00	0,00
2	97	0,00	0,07
3	233	0,00	0,06
4	281	0,00	0,08
5	313	0,00	0,8
6	769	0,00	1,2
7	7481	0,01	6,9
8	12073	0,01	17,6
9	98953	0,02	596,4
10	2459489	0,04	4345,8

Tabela 4 – Testes para Modular Square Roots

Conforme os autores da abordagem afirmam em seu trabalho, testar se um elemento é resíduo quadrático ou não quadrático é bastante complexo para grandes números primos. Esta tarefa está diretamente ligada com o tempo de espera para a solução do quebra-cabeça. A situação fica ainda mais demorada quando o primeiro número escolhido acaba sendo não-quadrático. Isto significa que é necessário decrementar este número por 1 e testar novamente todas as possibilidades de elementos dentro do conjunto.

Através dos resultados foi possível perceber que números primos pequenos não fornecem muita dificuldade para o cliente. Nestes casos, mesmo que a solução do quebra-cabeça estivesse correta, o servidor provavelmente negaria o acesso a seus serviços, solicitando ao cliente que escolhesse um número primo maior para o quebra-cabeça e tentasse novamente.

Os testes com números primos com 30 *bits* demoraram algumas horas para serem solucionados e, ao tentar rodar um teste utilizando um número primo de 512 bits, não houve recurso de memória suficiente. Dito isto, é possível afirmar a dificuldade do problema proposto, bem como comprovar a eficácia deste para cenários de negação de serviço, onde o objetivo do mesmo é fazer com que o cliente realize um grande esforço computacional, medindo este esforço pelo tamanho do número primo escolhido.

8 CONSIDERAÇÕES FINAIS

Este trabalho teve como objetivo o desenvolvimento de um programa que implementasse três algoritmos de quebra-cabeça criptográfico estudados. Antes de iniciar a programação, foi feito um longo estudo sobre o cenário atual de segurança em computação, avaliando a importância de compreender e utilizar mecanismos como os que foram aqui explicados. Após a finalização do programa, foi possível atestar a eficácia de cada um deles através de resultados experimentais obtidos com rotinas de testes bem definidas para cada abordagem.

Além de detalhar todos os aspectos matemáticos, práticos e conceituais dos três algoritmos escolhidos, foi exposto também diversos outros protocolos de quebra-cabeça criptográfico existentes, levando em conta as características de cada um deles e a aplicabilidade dentro do vasto ambiente de segurança e sigilo da informação nos dias de hoje.

Nos capítulos 5 e 6 podemos observar como foi organizado todo o projeto do código e como as funções principais funcionam, evidenciando os métodos de criação, solução e verificação de resultado. Além disso, observações sobre os testes unitários realizados foram feitas, comprovando que todos estão funcionando de acordo com o esperado. Ficou claro que as características apresentadas para cada implementação são bastante diferentes, e a escolha por utilizar qualquer um desses métodos para reforçar os requisitos de segurança deve ser baseada nas necessidades específicas de cada caso.

Foi possível concluir, analisando a pesquisa feita em cima das teorias de quebra-cabeças criptográficos existentes, que esses mecanismos são realmente importantes, visto que podem ser aplicados em diferentes conceitos e, portanto, podem auxiliar no reforço da segurança em diversas situações diferentes, desde envio de mensagens no futuro até evitar ataques de negação de serviço. Quanto mais mecanismos voltados para a proteção contra ataques maliciosos existirem, melhor.

Além disto, analisando os testes executados em cada uma das abordagens implementadas, foi possível perceber a eficiência dos algoritmos, que funcionam corretamente de acordo com o escopo no qual foram criados. Em uma breve comparação podemos concluir que cada um possui vantagens dependendo do contexto em que se deseja aplicar.

Por exemplo, em um contexto onde a prioridade é ter pouco custo por parte do servidor, as abordagens *Subset Sum* e *Modular Square Roots* são mais adequadas, enquanto em cenários onde a precisão é prioritária, a escolha pelo *Time Lock* deve ser considerada a melhor.

Por fim, tendo em mente o exposto acima quanto à importância de quebra-cabeças criptográficos (bem como quaisquer outros meios de proteger-se), é coerente afirmar que a contribuição deste trabalho é, também, importante dentro deste âmbito. Quanto mais materiais, fontes, exemplos e códigos existirem implementando esse tipo de mecanismo, mais os usuários terão ferramentas disponíveis para aprender, entender e contribuir com o cenário atual.

8.1 TRABALHOS FUTUROS

Como já foi dito anteriormente neste trabalho, a área de segurança da computação é muito ampla, englobando diversos mecanismos. Assim como existem diferentes maneiras de se proteger, é possível combiná-las para reforçar o sigilo e a segurança das informações nos mais diferentes contextos existentes. Por se tratar de uma área com grande abrangência, existe muito que pode ser feito, abrindo um leque de possibilidades para trabalhos futuros.

Seria interessante, por motivos de aprimoramento do que já foi feito, disponibilizá-lo para utilização *online*, criando assim uma *interface* intuitiva e adequada para os usuários que quisessem testar. Fica como proposta também detalhar e implementar os outros tipos de quebra-cabeças estudados neste trabalho, mesmo aqueles que não trabalham com o fator tempo, pois a aplicabilidade destes é bem variada.

Uma das possibilidades para estender a pesquisa é implementar o protocolo como um todo. Este trabalho implementou e testou os conceitos matemáticos envolvidos nos algoritmos, comprovando que as abordagens funcionam corretamente em todas as suas fases (construção, solução e verificação do quebra-cabeça criptográfico). Contudo, uma limitação deste trabalho está no fato de que a troca de mensagens entre servidor (parte que gera e verifica o problema) e o cliente (parte que resolve o problema) não foi implementada. É possível testar as três fases do quebra-cabeça dentro da mesma máquina apenas. Da maneira

como foi implementado, ainda não é possível que uma máquina gere o problema e envie para outra resolver.

O desafio que fica para futuros trabalhos é compreender como, em tempo de geração do quebra-cabeça, a máquina que funcionará como servidor terá conhecimento da capacidade de processamento do cliente (quantas operações de quadrado modular ele faz por segundo), para que possa utilizar este valor para criar o problema de maneira que o cliente só resolva no tempo correto. Com a obtenção dessa informação, pode-se trabalhar na implementação do protocolo por inteiro, incluindo as trocas de mensagens entre as máquinas.

Em suma, as propostas para trabalhos futuros são as seguintes:

- Incluir interface intuitiva para o usuário;
- Disponibilizar para utilização *online*;
- Implementar os outros tipos de quebra-cabeças, mesmo aqueles que não trabalham com o fator tempo;
- Implementar o protocolo como um todo:
  - Construção e solução em máquinas diferentes;
  - Protocolo de troca de mensagem entre máquinas;
  - Descobrir poder de processamento da máquina cliente de maneira eficaz;

REFERÊNCIAS

ATOM. *Atom*. 2015. <https://atom.io/>. Acessado em 15/10/2017.

AURA, T.; NIKANDER, P.; LEIWO, J. Dos-resistant authentication with client puzzles. Security Protocols Workshop, p. 170–181, 2000.

BAJAJ, R. D.; GOKHALE, U. M. Aes algorithm for encryption. International Journal of Latest Research in Engineering and Technology (IJLRET), v. 02, p. 63–68, 2016.

BARROS, C. F. D. *Introdução à Teoria dos Números*. Rio de Janeiro, Brasil, 2008.

BRAINARD, J.; JUELS, A. Client puzzles: A cryptographic defense against connection depletion attacks. *The 1999 Network and Distributed System Security Symposium*, p. 151–165, 1999.

CAMPELLO, A. *Reticulados, Teoremas de Minkowski Revisitados e Teoremas de Somas de Quadrados*. São Paulo, Brasil, 2014.

DAEMEN, J.; RIJMEN, V. Aes proposal: Rijndael. In: *The Rijndael Block Cipher*. [S.l.: s.n.], 1999. p. 1–45.

DIAS, M. L. *Introdução à Teoria dos Reticulados e Reticulados de Subgrupos*. Tese (Doutorado) — Universidade Federal de Campina Grande, Paraíba, Brasil, 2013.

DIFFIE, W.; HELLMAN, M. E. New directions in cryptography. IEEE transactions on information theory, v. 22, n. 6, p. 644–654, 1976.

FENG, W. et al. The design and implementation of network layer puzzles. In: IEEE. [S.l.], 2005.

GAO, Y. *Efficient Trapdoor-Based Client Puzzle System against DoS Attacks*. Tese (Doutorado) — School of Information Technology and Computer Science, University of Wollongong, Wollongong, Australia, 2005.

GARAY, J. A.; POMERANCE, C. Timed fair exchange of standard signatures. LNCS, v. 2742, p. 190–207, 2003.

GROZA, B.; PETRICA, D. On chained cryptographic puzzles. 3rd Romanian- Hungarian Joint Symposium on Applied Computational Intelligence, p. 25–26, 2006.

HOFFSTEIN, J.; PIPHER, J.; SILVERMAN, J. H. An introduction to mathematical cryptography. Springer Science+Business Media, LLC, New York, Estados Unidos, p. 524, 2008.

JERSCHOW, Y. I.; MAUVE, M. Offline submission with rsa time-lock puzzles. In: IEEE. *10th IEEE International Conference on Computer and Information Technology (CIT)*. [S.l.], 2010. p. 1058–1064.

JERSCHOW, Y. I.; MAUVE, M. Non-parallelizable and non-interactive client puzzles from modular square roots. In: IEEE. *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*. [S.l.], 2011. p. 135–142.

LEHMER, D. H. Computer technology applied to the theory of numbers. Studies in Number Theory, p. 117–151, 1969.

LENSTRA, A. K.; LENSTRA, H. K.; LOVÁSZ, L. Factoring polynomials with rational coefficients. Mathematische Annalen, p. 515–534, 1982.

MA, M. Mitigating denial of service attacks with password puzzles. International Conference on Information Technology: Coding and Computing, p. 621–626, 2005.

MERKLE, R. C. Secure communications over insecure channels. *Communications of the ACM*, ACM, v. 21, n. 4, p. 294–299, 1978.

MICROSOFT. *Microsoft*. 1975. <https://www.microsoft.com>. Acessado em 15/10/2017.

National Institute of Standards and Technology (NIST). *FIPS 180-4 - Secure Hash Standard (SHS)*. Gaithersburg, MD: [s.n.], 2015.

NIST. *National Institute of Standards and Technology*. 1901. <https://www.nist.gov/>. Acessado em 24/10/2017.

PYTHON. *Python Software Foundation*. 2001. <https://www.python.org/>. Acessado em 15/10/2017.

RADZISZOWSKI, S. P.; KREHER, D. L. Solving subset sum problems with l3 algorithm. Journal of Combinatorial Mathematics and Combinatorial Computing, p. 49–63, 1988.

RIVEST, R. L. *The MD6 hash function*. Massachusetts, Estados Unidos, 2008.

RIVEST, R. L.; SHAMIR, A.; WAGNER, D. A. Time-lock puzzles and timed-release crypto. Massachusetts Institute of Technology, 1996.

SAGE. *System for Algebra and Geometry Experimentation*. 2005. <http://www.sagemath.org/>. Acessado em 23/10/2017.

SCHNEIER, B. *Applied cryptography: protocols, algorithms, and source code in C*. [S.l.]: john wiley & sons, 2007.

STUTTARD, D.; PINTO, M. *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws*. Indiana, Estados Unidos: John Wiley & Sons, 2011. 912 p.

SYVERSON, P. F. Weakly secret bit commitment: Applications to lotteries and fair exchange. In: IEEE. *Computer Security Foundations Workshop, proceedings 11th*. [S.l.], 1998.

TRITILANUNT, S. et al. Toward non-parallelizable client puzzles. In: SPRINGER. *CANS*. [S.l.], 2007. p. 247–264.

WATERS, B. et al. New client puzzle outsourcing techniques for dos resistance. The 11th ACM Conference on Computer and Communications Security, p. 246–256, 2004.

XINYUE, D. *An Introduction to Lenstra-Lenstra-Lovasz Lattice Basis Reduction Algorithm*. Massachusetts, Estados Unidos, 2016.

ANEXO A – Código

A seguir, temos o código desenvolvido para cada uma das três abordagens estudadas neste trabalho. Iniciamos pela apresentação da classe *TimeLock.py*. Na sequência encontramos, na respectiva ordem, as classes *SubsetSum.py* e *ModularSquareRoots.sagews*. Para cada abordagem será também exposta a rotina de testes criada.

A.1 CÓDIGO PARA TIME LOCK

```
1  import os, random, time
2  from random import SystemRandom
3
4  from cryptography.hazmat.backends               import default_backend
5  from cryptography.hazmat.primitives.asymmetric  import rsa
6  from cryptography.hazmat.primitives.ciphers     import Cipher, algorithms, modes
7
8  class TimeLockPuzzle:
9
10     def __init__(self, message, seconds, keysize=2048):
11
12         rsa_   = rsa.generate_private_key(
13             public_exponent = 65537,
14             key_size        = keysize,
15             backend         = default_backend()
16         )
17         self.n = rsa_.private_numbers().public_numbers.n
18
19         totalsquares = 0
20         for num in range(0,25):
21             numbern = rsa_.private_numbers().public_numbers.n
22             numbera = random.SystemRandom().randint(1, numbern)
23             spersecond = self.compute_s(numbera, numbern)
24             totalsquares = totalsquares + spersecond
25
26         totalsquares = totalsquares/25
27
28         self.M = bytes(message)
29         self.T = int(seconds)
30         self.S = int(totalsquares)
31
32         p = rsa_.private_numbers()._p
33         q = rsa_.private_numbers()._q
34
35         self.phi = (p - 1) * (q - 1)
36         self.t = self.T * self.S
37         self.K = self.gen_random_n_digits(16)
38
```

```
39      def __exit__(self):
40          self.n = None
41          self.K = None
42
43      def gen_random_n_digits(self, n):
44          cryptogen = SystemRandom()
45
46          def _gen():
47              return cryptogen.randint(0000000000000000, 9999999999999999)
48          random = _gen()
49          while len(str(random)) != n:
50              random = _gen()
51          return random
52
53      def create_puzzle(self):
54          self.a  = random.SystemRandom().randint(1, self.n)
55          self.iv = os.urandom(16)
56          k_bytes = self.to_bytes(self.K)
57          enc = Cipher(
58              algorithms.AES(k_bytes),
59              modes.OFB(self.iv),
60              backend = default_backend()
61          ).encryptor()
62          self.C_M = enc.update(self.M) + enc.finalize()
63
64          e = pow(2, self.t, self.phi)
65          b = pow(self.a, e, self.n)
66          self.C_K = self.K + b
67          return self.C_M, self.iv, self.C_K, self.a, self.t, self.n
68
69      def to_bytes(self, K):
70          x = [int(i) for i in str(K)]
71          return bytes(x)
72
73      def solve(self, iv, message, C_K, a, t, n):
74          b = self.compute_b(a, t, n)
75          key = C_K - b
76          dec = Cipher(
77              algorithms.AES(self.to_bytes(key)),
78              modes.OFB(iv),
79              backend=default_backend()
80          ).decryptor()
81          return dec.update(message) + dec.finalize()
82
83      def compute_b(self, a, t, n):
84          return pow(a, pow(2, t), n)
85
86      def compute_s(self, a, n):
87          start_time = time.time()
88          num_squares = 100000
89          for num in range(0, num_squares):
```

```
90              b = pow(a, 2, n)
91          end_time = time.time()
92          tot_time = end_time-start_time
93          return num_squares/tot_time
94
95      def check(self, original_message, found_message):
96          return original_message == found_message
```

A.2 TESTE PARA TIME LOCK

```
1   totalseconds = 1
2   for num in range(0,20):
3       original_message = b'Universidade Federal de Santa Catarina'
4
5       timelock = TimeLockPuzzle(
6           message=original_message,
7           seconds=totalseconds
8       )
9       startconstruction_time = time.time()
10      C_M, iv, C_K, a, t, n = timelock.create_puzzle()
11      endconstruction_time = time.time()
12      construction_time = endconstruction_time - startconstruction_time
13
14      startsolution_time = time.time()
15      found_message = timelock.solve(iv, C_M, C_K, a, t, n)
16      endsolution_time = time.time()
17      solution_time = endsolution_time - startsolution_time
18
19      print ("Construction time: ",construction_time)
20      print ("Solution time: ",solution_time)
21
22      totalseconds = totalseconds+1
23
24      ok = timelock.check(original_message, found_message)
25      self.assertTrue(ok)
```

A.3 CÓDIGO PARA SUBSET SUM

```
1   import hashlib
2   from random import SystemRandom
3
4   from L3 import create_matrix_from_knapsack, best_vect_knapsack, lll_reduction, \
5       create_matrix_from_knapsack_orig
6
```

```
7   class SubSetSum:
8
9       def __init__(self, K):
10          self.cryptogen = SystemRandom()
11          self.K = K
12
13          self.IDr = self.cryptogen.randint(1000,999999)
14          self.NIr = self.cryptogen.randint(1000,999999)
15
16      def create_puzzle(self, IDi, Ni):
17          self.S = self.cryptogen.randint(1000, 999999)
18
19          hash = self.get_hash(IDi, Ni, self.NIr, self.IDr, self.S)
20          hash = bin(int(hash, 16))[2:].zfill(8)
21
22          self.C = hash[-self.K:]
23          self.C = [int(i) for i in self.C]
24
25          w = self.get_w(self.K)
26
27          W = 0
28          for idx, val in enumerate(self.C):
29              W += self.C[idx] * w[idx]
30
31          return w[0], W, self.K
32
33      def check(self, solution):
34          return solution == self.C
35
36      def get_hash(self, *args):
37          m = hashlib.sha256()
38          for a in args:
39              m.update(self.to_bytes(a))
40          return m.hexdigest()
41
42      def solve(self, wi, W, k):
43          w = [wi]
44          i = 0
45          while i < k - 1:
46              w.append(int(self.get_hash(w[i]), 16))
47              i += 1
48
49          matrix = create_matrix_from_knapsack_orig(w, W)
50
51          reduced = lll_reduction(matrix)
52          solution = best_vect_knapsack(reduced)
53
54          return solution
55
56
57      def to_bytes(self, K):
```

```
58          x = [int(i) for i in str(K)]
59          return bytes(x)
60
61      def get_w(self, k):
62          w = [self.cryptogen.randint(1000, 999999)]
63          i = 0
64          while i < k - 1:
65              w.append(int(self.get_hash(w[i]), 16))
66              i += 1
67          return w
```

A.4 TESTE PARA SUBSET SUM

```
1   totaldifficulty = 10
2   for num in range(0,1):
3       subsetSum = SubSetSum(totaldifficulty)
4
5       startconstruction_time = time.time()
6       w, W, K = subsetSum.create_puzzle(11, 153)
7       endconstruction_time = time.time()
8       construction_time = endconstruction_time - startconstruction_time
9
10      startsolution_time = time.time()
11      solution = subsetSum.solve(w, W, K)
12      endsolution_time = time.time()
13      solution_time = endsolution_time - startsolution_time
14
15      print ("Puzzle's difficulty: ",totaldifficulty)
16      print ("Construction time: ",construction_time)
17      print ("Solution time: ",solution_time)
18
19      totaldifficulty = totaldifficulty+1
20
21      ok = subsetSum.check(solution)
22      self.assertTrue(ok)
```

A.5 CÓDIGO PARA MODULAR SQUARE ROOTS

```
1   import hashlib
2   import binascii
3   import math
4   import os
5   import random
6   import time
```

```
7   from Crypto.Util import number
8
9   def get_n_bit_prime(n):
10      p = 0
11      while (1 != p%8):
12          p = random_prime(2^n)
13      return p
14
15  def number_of_bits(n):
16      return int(math.log(n, 2)) + 1
17
18  def sha256(message):
19      sha = hashlib.sha256()
20      sha.update(message)
21      return sha.hexdigest()
22
23  def get_d(message, n):
24      c = ceil(n/256)
25      ret = ''
26      prev = message
27      for i in range(0, c):
28          prev = sha256(prev)
29          ret += prev
30      return long(ret, 16)
31
32  def is_prime(a):
33      return all(a % i for i in xrange(2, a))
34
35  def factorize(n):
36      factors = []
37
38      p = 2
39      while True:
40          while(n % p == 0 and n > 0):
41              factors.append(p)
42              n = n / p
43          p += 1
44          if p > n / p:
45              break
46      if n > 1:
47          factors.append(n)
48      return factors
49
50  def calculate_legendre(a, p):
51      if a >= p or a < 0:
52          return calculate_legendre(a % p, p)
53      elif a == 0 or a == 1:
54          return a
55      elif a == 2:
56          if p % 8 == 1 or p % 8 == 7:
57              return 1
```

```
58          else:
59              return -1
60      elif a == p-1:
61          if p % 4 == 1:
62              return 1
63          else:
64              return -1
65      elif not is_prime(a):
66          factors = factorize(a)
67          product = 1
68          for pi in factors:
69              product *= calculate_legendre(pi, p)
70          return product
71      else:
72          if ((p-1)/2) % 2 == 0 or ((a-1)/2) % 2 == 0:
73              return calculate_legendre(p, a)
74          else:
75              return (-1)*calculate_legendre(p, a)
76
77  def find_b(a, p):
78      set_of_b = []
79      for b in range(1, p):
80          possible_b = b**2 - 4*a
81          nonresidue = calculate_legendre(possible_b, p)
82          if nonresidue == -1:
83              set_of_b.append(b)
84      return set_of_b
85
86  def cipolla_lehmer(a, p):
87      b = find_b(a, p)
88      max_list = len(b)-1
89      list_position = randint(0,max_list)
90      b = b[list_position]
91
92      F.<x> = GF(p)[]
93
94      f = x**2 -b*x + a
95      r = x**( (p+1)/2 ) % f
96
97      return r
98
99  def create_puzzle(message, nbits):
100     p = get_n_bit_prime(nbits)
101     n = number_of_bits(p)
102
103     hash_final = get_d(message, n)
104     hash_bits = bin(hash_final)
105     first_bits = hash_bits[2:n+1]
106     d = int(first_bits, 2)
107
108     is_residue = calculate_legendre(d, p)
```

```
109     while is_residue != 1:
110         d = d-1
111         is_residue = calculate_legendre(d,p)
112 
113     return d, p
114 
115 def solve(a, p):
116     r = cipolla_lehmer(a, p)
117     return r
118 
119 def check(message, r, p):
120     n = number_of_bits(p)
121 
122     hash_final = get_d(message, n)
123     hash_bits = bin(hash_final)
124     first_bits = hash_bits[2:n+1]
125     d = int(first_bits, 2)
126 
127     delta = 20
128     print d - (int(r)^2)%p < delta
```

A.6 TESTE PARA MODULAR SQUARE ROOTS

```
1  message = "Teste"
2  testnumber = 1
3  for nbits in range(5, 35):
4      startconstruction_time = time.time()
5      a, p = create_puzzle(message, nbits)
6      endconstruction_time = time.time()
7      construction_time = endconstruction_time - startconstruction_time
8  
9      startsolution_time = time.time()
10     r = solve(a, p)
11     endsolution_time = time.time()
12     solution_time = endsolution_time - startsolution_time
13 
14     check(message, r, p)
15 
16     print ("Test number: ",testnumber, " ------------------------------------")
17     print ("Prime number: ",p)
18     print ("Construction time: ",construction_time)
19     print ("Solution time: ",solution_time)
20 
21     testnumber = testnumber + 1
```

ANEXO B - Artigo

QUEBRA-CABEÇAS CRIPTOGRÁFICOS

Marina da Silva Coelho[1]

[1]Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Campus Universitário – Florianópolis – SC – Brasil

marina.s.coelho@grad.ufsc.br

Abstract. *It is common knowledge, within the area of Computer Security, that we have different mechanisms to guarantee the principles of confidentiality, integrity, availability and authenticity. One of the mechanisms known today is called Cryptographic Puzzle. This work intends to explore the different types of existing puzzles, as well as the difference between them and the applicability of each one in the area of security. In addition, this paper will detail, implement and test three puzzle approaches. Experimental results will allow us to confirm the efficiency of the approaches and to better understand the mathematical concepts involved. In addition, it will be possible to compare one approach with the other, considering its complexity, computational cost, and time precision.*

Resumo. *É de comum conhecimento, dentro da área de Segurança em Computação, que dispomos de diferentes mecanismos para garantir os princípios de confidencialidade, integridade, disponibilidade e autenticidade. Um dos mecanismos conhecidos hoje chama-se Quebra-Cabeça Criptográfico. Este trabalho pretende explorar os diferentes tipos de quebra cabeça existentes, bem como a diferença entre eles e a aplicabilidade de cada um na área de segurança. Além disso, este trabalho irá detalhar, implementar e testar três abordagens de quebra-cabeça. Os resultados experimentais encontrados nos permitirão confirmar a eficiência das abordagens e compreender melhor os conceitos matemáticos envolvidos. Além disso, será possível comparar uma abordagem com a outra no que diz respeito à sua complexidade, custo computacional e precisão de tempo.*

1. Introdução

Uma das maneiras de proteger-se contra diferentes tipos de ataque à segurança em computação chama-se Quebra-Cabeça Criptográfico, que pode ser visto como um problema que deve ser resolvido pelo usuário para poder obter acesso à alguma coisa. A primeira menção de um *Crypto Puzzle* foi feita por Ralph Merkle em seu trabalho *Secure communications over insecure channels* [Merkle 1978]. Segundo o autor, um quebra-cabeça pode ser definido como um criptograma que foi feito com o objetivo de ser quebrado. Assim como é possível cifrar um texto ao se produzir um criptograma com esses texto, deve ser possível proteger uma informação produzindo um quebra-cabeças. A diferença entre os dois conceitos é que um criptograma ideal não deve ser quebrado, enquanto o objetivo de um quebra-cabeças é, justamente, ser quebrado depois de um determinado esforço ou tempo.

Este tipo de mecanismo de segurança pode ser aplicado em diferentes contextos. A aplicação mais conhecida é em cenários onde trabalha-se com o tempo. Para estes

casos, o quebra-cabeça só pode ser quebrado depois de uma certa quantia de tempo, garantindo o sigilo da informação até o momento desejado [Rivest et al. 1996]. Para atender à este contexto, o quebra-cabeça criptográfico deve possuir uma propriedade chamada "não paralelização". Esta característica diz que o cliente (como chamamos a pessoa que quer solucionar o problema proposto) não deve ser capaz de acelerar a resolução do problema em questão através da distribuição do mesmo em vários processadores [Tritilanunt et al. 2007]. Esta propriedade garante que o problema não poderá ser resolvido antes do tempo previsto, fazendo com que seja um mecanismo extra de segurança para casos onde queremos que uma informação apenas se torne disponível após um certo período de tempo.

Estes quebra-cabeças criptográficos não paralelizáveis são aplicados em diversos contextos. Em seu trabalho *Weakly Secret Bit Commitment: Applications to Lotteries and Fair Exchange* [Syverson 1998], Paul Syverson fala sobre a utilização de quebra-cabeças para proteger o resultado de jogos de loteria. Neste cenário o resultado da loteria estaria basedo na lista de *tickets* vendidos, onde cada novo *ticket* vendido impactaria no resultado final da loteria. Uma vez encerrada as vendas, o tempo para computar o resultado da loteria deve ser suficientemente longo para que a lista final de *tickets* vendidos tenha sido publicada, para auditoria do resultado. Ao mesmo tempo deve ser suficientemente rápido para que ninguém possa calcular a saída da loteria antes do resultado oficial, através da utilização da lista publicada.

Tendo em mente a importância deste mecanismo, o foco deste trabalho está em compreender o cenário atual de quebra-cabeças criptográficos, conseguindo diferenciar cada uma das propostas existentes de acordo com a melhor aplicação que lhes cabe. Pretende-se fazer um estudo sobre algumas abordagens existentes, suas propriedades, vantagens e desvantagens. Além disso, este trabalho visa explicar detalhadamente três abordagens específicas que trabalham com o fator tempo, e implementá-las para a realização de uma série de testes, expondo também os resultados experimentais encontrados. Atrvés destes resultados será possível comparar uma abordagem com a outra no que diz respeito à sua complexidade, custo computacional e precisão de tempo.

O resto do trabalho está estruturado da seguinte maneira: No capítulo 2 são expostos alguns conceitos básicos, necessários para o entendimento do contexto matemático sobre o qual os quebra-cabeças se fundamentam. No capítulo 3 encontramos o detalhamento das três abordagens que foram implementadas, conhecidas como *Time Lock*, *Subset Sum* e *Modular Square Roots*. Já no capítulo 4 são expostos os resultados experimentais encontrados com o código produzido. Por fim, no capítulo 5 temos a conclusão e propostas de trabalhos futuros.

2. Conceitos básicos

Este capítulo apresenta três conceitos que são de extrema importância para o entendimento das abordagens de quebra-cabeça criptográficos que serão apresentadas, implementadas e testadas. Tais conceitos são: a função *hash*, na Seção 2.1, o problema da mochila, na Seção 2.2, o algoritmo de Lenstra-Lenstra-Lovász [Lenstra et al. 1982], na Seção 2.3, os conceitos matemáticos envolvidos na definição de resíduos quadráticos e não quadráticos, na Seção 2.4 e, por fim, as definições de complexidade (linear, polinomial e exponencial) na Seção 2.5.

2.1. Função Hash

Uma função de *hash* criptográfica h mapeia uma entrada M de tamanho variável de cadeia para uma *string* (texto) de comprimento fixo em *bits*. Para ser usada no contexto de segurança, a função *hash* deve contemplar algumas propriedades:

- Resistência a pré-imagem: deve ser inviável para um adversário, dado y, calcular M, tal que *hash*(M) = y;
- Resistência à colisão: deve ser inviável para um adversário encontrar valores M_1 e M_2, tal que *hash*(M_1) = *hash*(M_2);
- Resistência à segunda pré-imagem: deve ser inviável para um adversário, dado M_1, encontrar um valor diferente M_2, tal que *hash*(M_1) = *hash*(M_2);

2.2. Problema da mochila

Trata-se de um problema que tem como objetivo encontrar um subconjunto de valores, dado um conjunto de números nomeados "pesos", cuja soma resulte no maior número possível, desde que este não ultrapasse um valor S previamente estipulado [Tritilanunt et al. 2007]. Podemos pensar em S como sendo a capacidade total de peso que uma mochila pode carregar, e os diferentes números do conjunto recebido podem ser considerados itens, cuja soma de uma coleção destes itens não deve exceder a capacidade máxima da mochila, por isto o problema recebeu este nome. Um dos mecanismos existentes para solucionar o problema da mochila é o chamado algoritmo *LLL*, ou L^3, como veremos a seguir.

2.3. Algoritmo LLL

Antes de falarmos do algoritmo de Lenstra-Lenstra-Lovász [Lenstra et al. 1982], é necessário introduzir o conceito de reticulados. Em matemática, um reticulado é uma estrutura $L = (L, R)$ tal que L tem seus elementos parcialmente ordenados por uma relação R e, para cada dois elementos a, b no conjunto L, existe supremo (menor limite superior) e ínfimo (maior limite inferior) de a, b [Dias 2013].

Para prosseguir, é preciso também entender o conceito de vetores linearmente independentes. Seja V um espaço vetorial e $v_1, v_2, v_3, ..., v_n \in V$ alguns de seus vetores, considerando a equação:

$$a_1v_1 + a_2v_2 + a_3v_3 + ... + a_nv_n = 0$$

Se a única solução possível para esta equação for

$$a_1 = a_2 = a_3 = ... = a_n = 0$$

então dizemos que os vetores de V são Linearmente Independentes. Já se houver mais de uma solução para esta equação, dizemos que $v_1, v_2, v_3, ..., v_n$ são Linearmente Dependentes. Quando estamos tratando de reticulados, este conceito é necessário para entender a definição de base.

Se pensarmos em $m \leq n$ vetores linearmente independentes: $v_1, v_2, ..., v_m \in \mathbb{R}^n$, um reticulado Λ com base $(v_1, v_2, ..., v_m)$ é o conjunto de todas as combinações lineares inteiras de v_i, onde $1 \leq i \leq m$ [Campello 2014], o que significa:

$$\Lambda = \{u_1v_1 + ... + u_mv_m : u_1, ..., u_m \in \mathbb{Z}\}$$

O conjunto $\{v_1, v_2, ..., v_m\}$ então é denominado uma base de Λ. O uso de reticulados é frequente dentro da matemática e da criptografia, sendo o problema do vetor mais curto (conhecido como *SVP - Shortest vector problem*) o problema mais famoso e mais estudado. Ele consiste em encontrar o vetor mais curto dentro de um reticulado [Xinyue 2016]. Para encontrar o vetor mais curto é necessário fazer uma redução de base, que é um processo utilizado para reduzir uma base B de um reticulado, substituindo-a por uma base B' com vetores de menor tamanho (norma), sem que isso mude o reticulado. Utilizamos o algoritmo LLL para fazer esta redução de maneira eficiente. Este algoritmo é utilizado pois, segundo [Tritilanunt et al. 2007], é o método mais eficiente conhecido para encontrar vetores pequenos.

2.4. Resíduos

Para entender os conceitos relacionados a resíduos quadráticos e não quadráticos, é necessário fazer uma revisão de como extrair raízes quadradas em módulo primo. Se tivermos um número primo p, e um número inteiro a em $\mathbb{Z}_p^*$, onde $1 \leq a \leq p-1$, então a solução da congruência $x^2 \equiv a \pmod{p}$ é chamada de raiz quadrada em módulo p. A solução pode ser inexistente ou, se existir, serão duas soluções x e $-x$. No primeiro caso, onde existe solução, *a* é chamado de resíduo não quadrático, enquanto no segundo, *a* é chamado de resíduo quadrático em módulo p [Jerschow and Mauve 2011]. Um resíduo quadrático é um elemento que é equivalente ao quadrado de algum outro elemento em $\mathbb{Z}_p^*$ [Barros 2008].

Por exemplo, em $\mathbb{Z}_7^*$:

$$1^2 \equiv 1 \pmod{7}$$

$$2^2 \equiv 4 \pmod{7}$$

$$3^2 \equiv 9 \equiv 2 \pmod{7}$$

$$4^2 \equiv 16 \equiv 2 \pmod{7}$$

$$5^2 \equiv 25 \equiv 4 \pmod{7}$$

$$6^2 \equiv 36 \equiv 1 \pmod{7}$$

É possível afirmar, através das equações acima, que 1, 2 e 4 são resíduos quadráticos em $\mathbb{Z}_7^*$, enquanto os outros elementos do conjunto $\{3, 5, 6\}$ são resíduos não quadráticos em módulo 7, pois não são equivalentes ao quadrado de nenhum outro elemento do conjunto. Para dizer se a é, ou não, um resíduo quadrático, é utilizado o símbolo de Legendre, denotado como $\left(\frac{a}{p}\right)$. O símbolo de Legendre é uma função cujo valor é 1 se a é um resíduo quadrático módulo p, -1 se é um resíduo não quadrático módulo p e 0 se $a = 0$.

2.5. Complexidade

Os algoritmos estudados serão analisados de acordo com uma série de propriedades, sendo uma delas a complexidade de cada um deles, relacionada ao tempo que cada um leva para realizar as operações exigidas na respectiva abordagem. Para os mecanismos de quebra-cabeças criptográficos estudados, precisamos entender três medidas diferentes de complexidade:

- Complexidade linear: melhor caso possível, onde o tempo de execução (medido como *O(n)*) aumenta linearmente de acordo com o tamanho da entrada, possibilitando um controle muito pontual do tempo de resolução do problema matemático envolvido;
- Complexidade polinomial: onde o tempo de execução é delimitado por uma expressão polinomial (*T(n)* = $O(n^k)$) para um número *k* fixo. Embora um tempo linear também possa ser polinomial, neste contexto temos a certeza de que, em casos onde o tempo é polinomial, não há um controle pontual do tempo de resolução do problema, pois as saídas variam de maneira mais abrupta dependendo da entrada fornecida;
- Complexidade exponencial: onde o tempo de execução é delimitado por (*T(n)* = $O(a^n)$) para $a > 1$ fixo. Neste caso o controle do tempo de resolução também não é pontual;

3. Abordagens estudadas

Para o seu pleno uso, quebra-cabeças criptográficos devem possuir uma série de propriedades. É difícil conseguir alcançar todas as características existentes, portanto é importante ter em mente qual a finalidade do quebra-cabeça. Para melhor entendermos estas propriedades, é necessário definir os conceitos de servidor e cliente. Dá-se o nome de servidor à parte que constrói o quebra-cabeça. Dá-se o nome de cliente à parte que irá solucionar o quebra-cabeça. Algumas das propriedades desejáveis foram estudadas neste trabalho. São elas:

- Unidade de Trabalho: Tipo das operações requeridas ao cliente para a solução do problema (por exemplo: operações de quadrado modular);
- Custo do Servidor: Esforço computacional da parte que cria o quebra-cabeça, medindo o custo em todas as fases do problema que dizem respeito ao servidor, desde sua pré-computação (nos casos em que houver), construção e verificação do problema;
- Custo do Cliente: Esforço computacional da parte que resolve o quebra-cabeça, na hora de receber, solucionar e enviar a resposta de volta ao servidor. Assume-se que cliente e servidor tenham capacidade de processamento e memória semelhantes;
- Não Paralelização: Capacidade de evitar que a parte que irá solucionar o quebra-cabeça possa fazê-lo de maneira mais rápida através da paralelização do trabalho em diferentes máquinas;
- Complexidade: Complexidade de cada algoritmo, com base no tempo, conforme visto anteriormente;

Das abordagens estudadas, foram selecionadas três que possuíam a propriedade de "não paralelização", desejável em cenários relacionados com o tempo. Estas três foram escolhidas para serem implementadas e testadas e serão apresentadas em detalhes neste capítulo. É importante ressaltar que estas não são as únicas abordagens que contemplam a propriedade de "não paralelização", mas foram selecionadas por serem amplamente conhecidas dentro da área de segurança da computação.

3.1. Time Lock

Ao construir um *Time Lock Puzzle*, o servidor quer cifrar uma mensagem M e mantê-la cifrada por um período de tempo T. Para isso, o servidor deve determinar quantas

operações de quadrado o cliente terá que executar para este período de tempo T. Para tal, o servidor obtém do cliente a sua capacidade de operações de quadrado por segundo. Essa quantidade é o valor S. Uma vez conhecendo S, o servidor determina o valor de t, isto é, $t = TS$.

Feito isso, o servidor procede a construção do quebra-cabeças propriamente dito. O primeiro passo é gerar dois números primos grandes p e q, escolhidos aleatoriamente, e computar o valor de n, tal que $n = pq$. Agora o servidor deve escolher uma chave K grande e cifrar a mensagem utilizando um cifrador simétrico qualquer, por exemplo, AES [Bajaj and Gokhale 2016].

$$C_M = AES(K, M) \tag{1}$$

Em seguida o servidor escolhe um valor a aleatoriamente, coprimo a n, tal que $1 < a < n$ e determina

$$b = a^{2^t} \mod n \tag{2}$$

A Equação 2 é o nosso quebra-cabeças. Resolver o quebra-cabeças significa determinar o valor de b. Como veremos, o servidor consegue resolver o quebra-cabeças de forma muito eficiente, diferentemente do cliente, que terá que fazer muitas operações de quadrado para resolvê-lo. A resolução do quebra-cabeças (Equação 2) é muito custosa para valores de 2^t muito maiores que n. Entretanto, existe uma simplificação que é possível ser feita pelo servidor, usando o Teorema de Euler:

$$a^{\phi(n)} \equiv 1 \pmod n$$

onde a é coprimo a n, consegue-se reduzir o expoente 2^t, ou seja, faz-se $2^t \pmod{\phi(n)}$. Para calcular o Totiente de Euler, o servidor faz:

$$\phi(n) = (p-1)(q-1) \tag{3}$$

O próximo passo deve ser a determinação do valor C_K, correpondente ao cifrамento da chave K. Para isso, o servidor resolve o quebra-cabeças, adicionando o resultado a K, ou seja

$$C_K = K + b \mod n \tag{4}$$

Para chegar a este valor de maneira rápida, ele faz:

$$e = 2^t \mod \phi(n) \tag{5}$$

$$b = a^e \mod n \tag{6}$$

O resultado disto é um quebra-cabeça que contém os parâmetros (n, a, t, C_K, C_M), onde C_M é a cifra da mensagem, que poderá ser decifrada com a utilização da chave K, que só será descoberta depois de resolver o quebra-cabeça. Qualquer outra variável utilizada durante a geração do problema (como os primos p e q) é destruída.

Quando o cliente recebe estes valores, é necessário resolver o quebra-cabeça e encontrar a chave K que cifrou a mensagem M. Para resolver o quebra-cabeças, o cliente deve computar b através da Equação 7.

$$b = a^{2^t} \mod n \tag{7}$$

Com o valor de *b* em mãos, é necessário subtrair este valor do parâmetro C_K recebido. O resultado desta subtração é a chave K a ser utilizada para decifrar C_M e assim encontrar M.

3.2. Subset Sum

Para que o método seja melhor compreendido, é necessário detalhar todas as variáveis que serão utilizadas:

- ID_I: Identificação do cliente;
- ID_R: Identificação do servidor;
- N_I: *Nonce* (número aleatório utilizado uma única vez) do cliente;
- N_R: *Nonce* (número aleatório utilizado uma única vez) do servidor;
- S: Parâmetro único e secreto escolhido para estabelecimento de comunicação;
- K: Dificuldade do problema;
- W: Limiar do quebra-cabeça;
- w_i: Peso do item i;
- n: Quantidade de itens do problema;
- $H()$: Operação *hash*;
- *Least Significant Bits* - $LSB(\alpha, K)$: Operação que obtém os K bits menos significativos de α;
- $C = LSB(\alpha, K)$

Quando um cliente quiser ganhar acesso a um determinado serviço de um servidor, este deve solicitar um quebra-cabeça, enviando junto à solicitação suas credenciais de identificação (ID_I, N_I). O servidor então vai armazenar as credenciais e escolher o parâmetro *S* de maneira aleatória. Este deve ser único, uma vez que irá identificar unicamente a comunicação entre cliente e servidor.

O próximo passo a ser executado pelo servidor é a escolha da dificuldade do problema. Quanto maior a dificuldade, mais tempo o cliente irá despender para solucionar o quebra-cabeça. Depois de escolher os valores de *S* e *K* o servidor vai fazer uma operação *hash* com os valores das cinco credenciais fornecidas como entrada $(ID_I, N_I, ID_R, N_R, S)$. O próximo passo é realizar a operação ***LSB*** para obter os K ***bits*** menos significantes do *hash* produzido. Esta operação dará origem à variável ***C***:

$$C = LSB(H(ID_I, N_I, ID_R, N_R, S), K)_2 \tag{8}$$

Depois disto é necessário gerar o limiar do quebra-cabeça e conjunto de itens. Para gerar os itens, primeiro escolhe-se o item w_1 aleatoriamente. Depois disso, os próximos itens são criados fazendo *hash* do item anterior. Já para gerar o limiar W devemos considerar os bits de *C*, que serão representados por variáveis que vão de C_i até C_k. Cada um destes bits será multiplicado pelo item w_i correspondente, conforme a Equação 9.

$$W = \sum_{i=1}^{k} C_i.w_i \tag{9}$$

O servidor irá então montar e enviar o quebra-cabeça, contendo os valores (w_1 (o primeiro item escolhido aleatoriamente),*W* (o peso máximo desejável) e *K* (dificuldade)), juntamente às credenciais do cliente e do servidor, para que o cliente possa verificar a autenticidade da comunicação.

O primeiro passo para o cliente, antes de tentar solucionar o problema proposto, é verificar as credenciais (ID_I, N_I, ID_R, N_R) enviadas pelo servidor junto com o quebra-cabeça, para checar se conferem com as credenciais enviadas anteriormente pelo cliente ao estabelecer conexão. O segundo passo a ser executado pelo cliente é a geração do conjunto de itens que será utilizado para a solução do problema. Para isto, o cliente obtém o parâmetro w_1, que representa o primeiro item da série de itens. Obtém também o valor *K*, que determina a quantidade de itens a serem gerados, e a partir daí pode obter todo o conjunto de itens subsequentes da mesma maneira que o servidor obteve: através de *hash* iterativo. Com o conjunto de itens na mão, o cliente pode iniciar o processo de resolução do quebra-cabeça.

O terceiro passo exige que o cliente construa um conjunto de bases B com os itens gerados, tal que:

$$b_1 = (1, 0, 0, ..., 0, w_1)$$
$$b_2 = (0, 1, 0, ..., 0, w_2)$$
$$b_3 = (0, 0, 1, ..., 0, w_3)$$
$$...$$
$$b_k = (0, 0, 0, ..., 1, w_k)$$
$$b_{k+1} = (0, 0, 0, ..., 0, -W)$$

É sobre este reticulado que será rodado o algoritmo *LLL*, buscando encontrar a melhor base possível [Hoffstein et al. 2008]. Segundo [Tritilanunt et al. 2007] o algoritmo em questão garante o retorno de um conjunto de vetores, onde o menor deve ser a solução para o quebra-cabeça. Para checar esta solução, o cliente deve multiplicar cada valor do vetor pelo item respectivo na lista de itens gerados, semelhante ao somatório feito pelo servidor em fase de construção. Este vetor pequeno que representa o resultado será chamado de *C'*. Para averiguar se este é o correto o cliente irá checar se:

$$W \stackrel{?}{=} \sum_{i=1}^{k} C'_i.w_i \tag{10}$$

Onde C'_i corresponde a cada um dos valores do menor vetor encontrado. O valor encontrado no somatório não deve ultrapassar o limiar W. Se o resultado satisfizer esta condição, o cliente envia de volta ao servidor uma série de valores: as credenciais de identificação (ID_I, N_I, ID_R, N_R), o quebra-cabeça recebido anteriormente (w_i, W, K) e o vetor *C'* encontrado.

3.3. Modular Square Roots

A abordagem explicada a seguir trata da técnica não interativa, onde o servidor não enviará o problema para o cliente. Ao invés disso, o próprio cliente irá construir o quebra-cabeça e solucioná-lo. Para isto, as duas partes (cliente e servidor) devem compartilhar previamente uma lista de números primos $p \equiv 1 \pmod 8$ com diferentes tamanhos em *bits*. Quando um cliente solicitar acesso a algum serviço de um servidor, este deve escolher um número primo da lista, de tamanho ***n bits***, e aplicar *hash* iterativo sobre a mensagem de requisição. O processo de fazer *hash* iterativo ocorrerá *c* vezes, onde *c* = $\lceil \frac{n}{k} \rceil$. O valor k corresponde ao tamanho do *hash* que será gerado. O resultado desta divisão é arredondado para o número inteiro imediatamente superior. O resultado deste *hash* iterativo será *d*, que são os primeiros ***n-1 bits*** encontrados como saída das operações realizadas, como exposto a seguir:

$$d = First_{n-1}(H(m)||H(H(m))||...||H^c(m)) \tag{11}$$

Onde $||$ representa a concatenação dos *hashs*. Agora é necessário verificar se d é um resíduo quadrático. Para isto, vai utilizar o símbolo de Legendre ($\frac{a}{p}$), onde p é o número primo escolhido inicialmente e $a = d$. Caso seja um resíduo não quadrático (resultado = -1), o cliente deve decrementar o valor de a até encontrar um resíduo quadrático (resultado = 1). Este resíduo a é adicionado ao quebra-cabeça, e já pode ser iniciada a fase de solução.

Os valores recebidos na fase de solução são d e r. Para utilizar a nomenclatura do artigo original, a variável d é chamada de a em fase de solução. Para solucionar o quebra-cabeça, é necessário encontrar o valor de r, que é a raiz quadrada de a em módulo p, ou seja, computar:

$$\sqrt{a} \equiv r \pmod p \tag{12}$$

Para isto é utilizado o algoritmo *Cipolla-Lehmer* [Lehmer 1969], que tem como entrada os valores d e p, e como saída as duas raízes quadradas de d em módulo p. Segundo [Jerschow and Mauve 2011], o algoritmo *Cipolla-Lehmer* funciona da maneira explicada a seguir:

Primeiro o símbolo de Legendre é computado. Se o resultado for -1, o algoritmo conclui que a é um resíduo não quadrático módulo p e termina sua execução. Caso o resultado seja 1, o algoritmo prossegue escolhendo um número inteiro b aleatoriamente dentro do conjunto $\mathbb{Z}_p^*$, tal que o resultado de $b^2 - 4a$ seja um resíduo não quadrático módulo p, ou seja:

$$(\frac{b^2 - 4a}{p}) = -1 \tag{13}$$

Tendo os valores de a e b, cria o polinômio $f(x) = x^2 - bx + a$ em $\mathbb{Z}_p^*[x]$ e computa:

$$r = x^{\frac{(p+1)}{2}} \pmod f \tag{14}$$

O valor r encontrado deverá ser um inteiro. Por fim, o algoritmo retorna r e o cliente o envia ao servidor, juntamente à posição do número primo escolhido na lista

compartilhada e a mensagem de requisição inicial. Desta maneira, não é preciso enviar explicitamente o número primo na comunicação.

Nesta abordagem, faz-se necessário detalhar também a fase de verificação, pois não se trata somente de averiguar se a solução enviada é igual à armazenada. Ao receber a requisição de acesso do cliente, bem como o quebra-cabeça resolvido (R), o servidor irá computar o valor d, da mesma maneira que o cliente computou, através da função de *hash* aplicada sobre a mensagem de requisição. Não é preciso utilizar o símbolo de Legendre para achar o resíduo quadrático a através do valor d encontrado, pois a probabilidade de $a = d$ é de 50%. Por consequência, a probabilidade de que $a = d-1$ é de 25% (pois, caso $a \neq d$, é provável que o algoritmo tenha decrementado o valor encontrado em 1, como vimos anteriormente). Para verificar se a solução encontrada está certa, o servidor utiliza o resultado r recebido e computa:

$$d - (r^2 \pmod p) < \delta \tag{15}$$

Onde δ é uma constante pequena (como, por exemplo, 20). Caso seja, de fato, menor que esta constante, a solução é considerada correta. Embora o resultado recebido esteja certo, o servidor não necessariamente vai aceitar a requisição de acesso do cliente, pois primeiro irá avaliar a dificuldade (tamanho) do número primo *p* escolhido. Se o servidor considerar o número primo pequeno, ele pode negar o pedido de acesso, e o cliente pode escolher um novo número primo *p* maior da lista para tentar novamente.

4. Resultados experimentais

Para cada uma das abordagens estudadas foi feito uma implementação. A aplicação foi feita na linguagem Python [Python 2001], em sua versão 3.5. Trata-se de uma linguagem funcional e de fácil sintaxe, trabalhando de modo bastante intuitivo com programadores que já têm experiência prévia em outras linguagens. É uma linguagem muito simples para produzir e rodar testes, sendo também de fácil, simples e rápida instalação em qualquer sistema operacional. Além disso, dispõe de bibliotecas muito boas que contém métodos importantíssimos para o contexto tratado, agilizando a programação em várias etapas.

Para a abordagem de *Modular Square Roots*, no entanto, o ambiente de desenvolvimento foi o *Sage* [Sage 2005], um software de matemática que possui recursos para operações com álgebra, combinatória, análise numérica, teoria dos números e cálculo. O software suporta a linguagem Python, na qual a abordagem foi desenvolvida. A preferência por este ambiente está no fato de que esta abordagem opera com polinômios sobre corpos finitos, e o Sage possui bibliotecas ótimas e de fácil utilização para cálculos com polinômios em anel.

Para cada abordagem implementada foram executados mais de 20 testes, porém, para tornar este capítulo mais objetivo, vamos expor os resultados encontrados em apenas uma parte dos testes. Para cada abordagem, serão mostrados os valores de, pelo menos, dez testes realizados. O resto deles seguiu o mesmo padrão. Todos foram executados em uma máquina 64 *bits* com sistema operacional Windows 8 [Microsoft 1975].

4.1. Time Lock

Os resultados experimentais encontrados para esta abordagem foram muito satisfatórios, visto que os valores foram muito aproximados, com a diferença de tempo girando entre 10 e 40 milésimos de segundos nos casos onde o tempo era muito pequeno. Já nos testes

Teste	Tempos [Segundos]		
	Fornecido	Construção	Solução
1	1	0,03	0,92
2	2	0,04	1,84
3	40	0,03	39,98
4	70	0,04	71,12
5	85	0,04	85,05
6	115	0,04	115,73
7	130	0,03	130,05
8	600	0,03	579,07
9	10.800	0,03	10.522,47
10	18.000	0,04	17.948,25

Tabela 1. Testes para Time Lock

em que a máquina ficou rodando por grandes quantias de tempo, a diferença foi um pouco maior. No caso em que o quebra-cabeça deveria ser desvendado em 3 horas, este foi solucionado 4 minutos antes, que foi a maior difeença encontrada, como podemos observar na Tabela 1. Já no caso onde o tempo fornecido foi 5 horas (18000 segundos), o erro foi de apenas 52 segundos. Ao analisar o tempo de geração do quebra-cabeça, os resultados também são ótimos, visto que seguem a regra fundamental que diz que o problema deve ser fácil de construir e difícil de resolver. Ao olharmos a coluna "Construção" observamos que nenhum quebra-cabeça demorou mais do que 0,04 milésimos de segundo para ser construído, não interessando se o parâmetro de entrada é 10 segundos, ou 18000. Para aumentar a precisão dos resultados encontrados, cada um dos testes foi executado mais de uma vez, produzindo também uma média do tempo de solução para cada caso. No maior dos casos testados, para 5 horas, a média retornada para 10 execuções deste teste foi 17.739,08.

4.2. Subset Sum

Nesta abordagem os resultados devem ser interpretados de maneira diferente, já que não há como prever o tempo exato em que serão solucionados. A característica forte deste protocolo é que o tempo para solucionar o quebra-cabeça deve aumentar na medida que a dificuldade fornecida nos testes aumenta. Iniciamos nossa rotina de teste com o valor de dificuldade 10 e aumentamos este até 65. Os valores de solução, expostos em segundos, são baixos, mas cresceram rapidamente. O resultados dos testes é mostrado na Tabela 4.3.

Teste	Tempos [Segundos]		
	Dificuldade	Construção	Solução
1	10	0,00	1,25
2	11	0,00	1,56
3	19	0,01	6,79
4	20	0,00	7,96
5	29	0,00	21,10
6	32	0,00	26,87
7	33	0,00	40,51
8	39	0,00	50,12
9	40	0,00	67,34
10	50	0,01	84,33
11	60	0,00	118,74

Tabela 2. Testes para Subset Sum

Os resultados também foram satisfatórios, visto que o quebra-cabeça obedece a proporção de aumento entre a dificuldade e o tempo. Já ao analisarmos a coluna "Construção", concluimos que a abordagem é ainda mais eficiente para o servidor do

que a anterior, visto que a geração do problema é quase instantânea, fazendo com que o servidor não tenha que desprender muitos recursos nesta fase.

A quantidade de tempo é bem pequena, porém segue a mesma linha da versão original criada por [Tritilanunt et al. 2007]. Ao testarem a própria abordagem os autores concluíram que para qualquer dificuldade menor que 25 o tempo de solução do quebra-cabeça era de 0.0 segundos, ou seja: instantâneo. Para casos onde a dificuldade era maior que 25 e menor que 50 o tempo ainda era muito baixo (existindo casos de solução em 0.1 segundos). Os autores só conseguiram um certo nível de dificuldade a partir do valor 50. Ainda segundo os autores da abordagem, não é recomendado usar $K > 100$, pois o tamanho da matriz de reticulado construída é grande demais, esgotando os recursos de memória.

4.3. Modular Square Roots

Já esta abordagem deve ser analisada de acordo com o número primo escolhido para geração do problema. Quanto maior o número, maior deve ser o tempo desprendido para a solução.

Teste	Tempos [Segundos]		
	Número primo	Construção	Solução
1	17	0,00	0,00
2	97	0,00	0,07
3	233	0,00	0,06
4	281	0,00	0,08
5	313	0,00	0,8
6	769	0,00	1,2
7	7481	0,01	6,9
8	12073	0,01	17,6
9	98953	0,02	596,4
10	2459489	0,04	4345,8

Tabela 3. Testes para Modular Square Roots

Assim como nas abordagens anteriores, podemos comprovar, através dos testes, que a aplicação funciona de acordo com o desejado. O tempo para solução do quebra-cabeça cresce de acordo com a dificuldade de encontrar a raíz quadrada de um elemento módulo um número primo. Isto acontece pois, quanto maior o número primo, maior a quantidade de elementos a serem testados dentro do corpo finito.

Conforme os autores da abordagem afirmam em seu trabalho, testar se um elemento é resíduo quadrático ou não quadrático é bastante complexo para grandes números primos do conjunto $p \equiv 1 \pmod 8$. Esta tarefa está diretamente ligada com o tempo de espera para a solução do quebra-cabeça. A situação fica ainda mais demorada quando o primeiro número escolhido acaba sendo não-quadrático. Isto significa que é necessário decrementar este número por 1 e testar novamente todas as possibilidades de elementos dentro do conjunto.

Os testes com números primos com 30 bits demoraram mais de horas para serem solucionados e, ao tentar rodar um teste utilizando um número primo $p \equiv 1 \pmod 8$ de 512 bits, não houve recurso de memória suficiente. Dito isto, é possível afirmar a dificuldade do problema proposto, bem como comprovar a eficácia deste para cenários de negação de serviço, onde o objetivo do mesmo é fazer com que o cliente realize um grande esforço computacional, medindo este esforço pelo tamanho do número primo escolhido.

5. Conclusão e Trabalhos futuros

Este trabalho teve como objetivo o desenvolvimento de um programa que implementasse três algoritmos de quebra-cabeça criptográfico estudados. Antes de iniciar a programação, foi feito um longo estudo sobre o cenário atual de segurança em computação, avaliando a importância de compreender e utilizar mecanismos como os que foram aqui explicados. Após a finalização do programa, foi possível atestar a eficácia de cada um deles através de resultados experimentais obtidos com rotinas de testes bem definidas para cada abordagem.

Analisando os testes executados em cada uma das abordagens implementadas, foi possível perceber a eficiência dos algoritmos, que funcionam corretamente de acordo com o escopo no qual foram criados. Em uma breve comparação podemos concluir que cada um possui vantagens dependendo do contexto em que se deseja aplicar. Por exemplo, em um contexto onde a prioridade é ter pouco custo por parte do servidor, as abordagens *Subset Sum* e *Modular Square Roots* são mais adequadas, enquanto em cenários onde a precisão é prioritária, a escolha pelo *Time Lock* deve ser considerada a melhor

Tendo em mente a importância de quebra-cabeças criptográficos no cenário de segurança em computação, é coerente afirmar que a contribuição deste trabalho é, também, importante dentro deste âmbito. Quando mais materiais, fontes, exemplos e códigos existirem implementando esse tipo de mecanismo, mais os usuários terão ferramentas disponíveis para aprender, entender e contribuir com o cenário atual.

Uma das possibilidades para estender a pesquisa é implementar o protocolo como um todo. Este trabalho implementou e testou os conceitos matemáticos envolvidos nos algoritmos, comprovando que as abordagens funcionam corretamente em todas as suas fases (construção, solução e verificação do quebra-cabeça criptográfico). Contudo, uma limitação deste trabalho está no fato de que a troca de mensagens entre servidor (parte que gera e verifica o problema) e o cliente (parte que resolve o problema) não foi implementada. É possível testar as três fases do quebra-cabeça dentro da mesma máquina apenas. Da maneira como foi implementado, ainda não é póssível que uma máquina gere o problema e envie para outra resolver.

O desafio que fica para futuros trabalhos é compreender como, em tempo de geração do quebra-cabeça, a máquina que funcionará como servidor terá conhecimento da capacidade de processamento do cliente (quantas operações de quadrado modular ele faz por segundo), para que possa utilizar este valor para criar o problema de maneira que o cliente só resolva no tempo correto. Com a obtenção dessa informação, pode-se trabalhar na implementação do protocolo por inteiro, incluindo as trocas de mensagens entre as máquinas.

Em suma, as propostas para trabalhos futuros são as seguintes:

- Incluir interface intuitiva para o usuário;
- Disponibilizar para utilização *online*;
- Implementar os outros tipos de quebra-cabeças, mesmo aqueles que não trabalham com o fator tempo;
- Implementar o protocolo como um todo:
  - Construção e solução em máquinas diferentes;
  - Protocolo de troca de mensagem entre máquinas;
  - Descobrir poder de processamento da máquina cliente de maneira eficaz;

Referências

Bajaj, R. D. and Gokhale, U. M. (2016). Aes algorithm for encryption. 02:63–68.

Barros, C. F. D. (2008). *Introdução à Teoria dos Números*. UFRJ, Rio de Janeiro, Brasil.

Campello, A. (2014). *Reticulados, Teoremas de Minkowski Revisitados e Teoremas de Somas de Quadrados*. Unicamp, São Paulo, Brasil.

Dias, M. L. (2013). *Introdução à Teoria dos Reticulados e Reticulados de Subgrupos*. PhD thesis, Universidade Federal de Campina Grande, Paraíba, Brasil.

Hoffstein, J., Pipher, J., and Silverman, J. H. (2008). An introduction to mathematical cryptography. page 524.

Jerschow, Y. I. and Mauve, M. (2011). Non-parallelizable and non-interactive client puzzles from modular square roots. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 135–142. IEEE.

Lehmer, D. H. (1969). Computer technology applied to the theory of numbers. pages 117–151.

Lenstra, A. K., Lenstra, H. K., and LovÁsz, L. (1982). Factoring polynomials with rational coefficients. pages 515–534.

Merkle, R. C. (1978). Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299.

Microsoft (1975). Microsoft.

Python (2001). Python software foundation.

Rivest, R. L., Shamir, A., and Wagner, D. A. (1996). Time-lock puzzles and timed-release crypto.

Sage (2005). System for algebra and geometry experimentation.

Syverson, P. F. (1998). Weakly secret bit commitment: Applications to lotteries and fair exchange. In *Computer Security Foundations Workshop, proceedings 11th*. IEEE.

Tritilanunt, S., Boyd, C., Foo, E., and Nieto, J. M. G. (2007). Toward non-parallelizable client puzzles. In *CANS*, pages 247–264. Springer.

Xinyue, D. (2016). *An Introduction to Lenstra-Lenstra-Lovasz Lattice Basis Reduction Algorithm*. Massachusetts Institute of Technology, Massachusetts, Estados Unidos.