HaaS - Hashcat como Serviço

Carlos de Souza Lima

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre no âmbito da dupla diplomação com a Universidade Tecnológica Federal do Paraná em Informática.

Orientadores:

José Carlos Rufino Amaro

André Koscianski

Esta versão da dissertação contempla as críticas e sugestões feitas pelo Júri.

Bragança

2023-2024

HaaS - Hashcat como Serviço

Carlos de Souza Lima

Dissertação apresentada à Escola Superior de Tecnologia e de Gestão de Bragança para obtenção do Grau de Mestre no âmbito da dupla diplomação com a Universidade Tecnológica Federal do Paraná em Informática.

Orientadores:

José Carlos Rufino Amaro

André Koscianski

Esta versão da dissertação contempla as críticas e sugestões feitas pelo Júri.

Bragança

2023-2024

Dedicatória

Dedico este trabalho a Deus, por me guiar e abençoar em cada passo desta jornada. Aos meus pais, Flavia e Carmelino, eterno alicerce de amor e apoio incondicional em cada passo da minha jornada acadêmica. A vocês, meu profundo reconhecimento pelo suporte emocional e financeiro que me impulsionaram a alcançar esta conquista. À minha esposa, Hellena, por sua compreensão infinita e paciência nos momentos de ausência, tornando cada reencontro um presente. E à minha irmã, Danielly, por sua amizade e apoio constante, sempre presente em todos os momentos da minha vida. Ao meu professor Adriano, que despertou em mim a paixão pela programação e tecnologia, plantando a semente que floresceu nesta realização. Aos meus amigos e familiares, por entenderem minhas ausências e me acolherem de braços abertos nos momentos de lazer e descontração. A todos vocês, meu eterno carinho e gratidão.

Agradecimentos

Agradeço profundamente ao meu orientador, Prof. José Carlos Rufino Amaro, pela sua dedicação incansável e orientação precisa ao longo deste mestrado. Sua disponibilidade, mesmo em seus momentos de descanso, e as reuniões semanais foram fundamentais. Sua paixão pela pesquisa e seu incentivo constante me inspiraram a buscar sempre o melhor.

Expresso minha sincera gratidão ao Rui Alves, cuja inspiração e orientação foram cruciais para a realização deste trabalho. Sua presença constante nas reuniões e sua solicitude em ajudar em todos os momentos foram inestimáveis. Agradeço especialmente por sua expertise no rOpenCL, ferramenta fundamental, e por sempre estar disponível para solucionar dúvidas e problemas. Sua generosidade e conhecimento foram verdadeiros presentes nesta jornada.

Agradeço ao meu coorientador, André Kosciansky, por suas valiosas revisões e insights que enriqueceram significativamente este trabalho. Sua experiência e olhar crítico foram essenciais para aprimorar a qualidade da pesquisa.

Agradeço à minha esposa, Hellena, por sua cuidadosa revisão do trabalho e suas valiosas sugestões que contribuíram para a clareza e qualidade do texto.

Agradeço à Universidade Tecnológica Federal do Paraná (UTFPR) e ao Instituto Politécnico de Bragança (IPB) pela oportunidade de realizar este mestrado e por proporcionarem um ambiente acadêmico estimulante e enriquecedor.

Por fim, agradeço a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho. Cada palavra de incentivo, cada ajuda e cada crítica construtiva foram importantes para me trazer até aqui.

Resumo

A crescente preocupação com a segurança cibernética impulsionou o desenvolvimento de ferramentas e técnicas para proteger dados confidenciais. O Hashcat, uma ferramenta de auditoria de senhas amplamente utilizada, é capaz de explorar o poder de processamento paralelo das GPUs para acelerar a quebra de *hashes* criptográficos. No entanto, a utilização eficiente do Hashcat em larga escala apresenta desafios, como a necessidade de gerenciar múltiplas GPUs e otimizar recursos computacionais.

Este trabalho apresenta o Hashcat como Serviço (HaaS), uma plataforma que visa simplificar e otimizar o processo de recuperação de senhas, tornando-o mais acessível e eficiente. O HaaS combina o Hashcat com tecnologias de containerização (Docker) e o *middleware remote OpenCL* (rOpenCL), permitindo a criação e gestão de instâncias do Hashcat em *containers*, capazes de tirar partido tanto de GPUs locais como remotas, e incluindo facilidades de migração das instâncias, de forma a maximizar o uso das GPUs.

A metodologia de desenvolvimento incluiu a experimentação com o Hashcat em diferentes ambientes, o projeto e desenvolvimento do HaaS e a avaliação do sistema desenvolvido. A avaliação do HaaS em diferentes configurações demonstrou resultados promissores, confirmando sua capacidade de lidar com cargas de trabalho intensivas e sua flexibilidade para se adaptar a diferentes cenários e disponibilidades de recursos.

Dessarte, o HaaS representa uma contribuição importante na área de recuperação de senhas, oferecendo uma plataforma escalável, eficiente e de fácil utilização.

Palavras-chave: Desenvolvimento Web, Criptografia Aplicada, Computação Paralela e Distribuída, Virtualização e Containerização, Sistemas Heterogêneos.

Abstract

The growing concern with cybersecurity has driven the development of tools and techniques to protect sensitive data. Hashcat, a widely used password auditing tool, is able to exploit the parallel processing power of GPUs to speed up the breaking of cryptographic hashes. However, the efficient use of Hashcat at large scale presents challenges, such as the need to manage multiple GPUs and optimize computational resources.

This work presents Hashcat as a Service (HaaS), a platform that aims to simplify and optimize the password recovery process, making it more accessible and efficient. HaaS combines Hashcat with container technologies (Docker) and remote OpenCL middleware (rOpenCL), allowing to create and manage Hashcat instances in containers, leveraging the processing power of local and remote GPUs, and supporting migration of instances in order to maximize GPU utilization.

The development methodology included experimentation with Hashcat in different environments, the design and development of HaaS and the evaluation of the developed system. The evaluation of HaaS in different configurations demonstrated promising results, confirming its ability to handle intensive workloads and its flexibility to adapt to different scenarios and resources availability.

As such, HaaS represents an valuable contribution in the password recovery field, offering a scalable, efficient and user-friendly platform.

Keywords: Web Development, Practical Cryptography, Parallel and Distributed Computing, Virtualization and Containerization, Heterogeneous Systems.

Conteúdo

Lista de Figuras

Siglas

CPU Unidade Central de Processamento. 14, 15, 50

CPUs Unidades Centrais de Processamento. 13

GPU Unidade de Processamento Gráfico. xii, xiii, 1–4, 11, 12, 14, 15, 26, 28, 29, 34–37, 39, 41–43, 49–54, 56–59, 62, 65–67, 69

GUI Interface Gráfica do Utilizador. 12

HaaS Hashcat como Serviço. 1–5, 17, 19, 20, 30, 31, 34–36, 38, 41, 49–53, 56, 65, 69

HTTP Protocolo de Transferência de Hipertexto. 22

IAM Gestão de Identidade e Acesso. 22

MD5 Algoritmo de Resumo de Mensagens 5. 8

REST Transferência de Estado Representacional. 22

SHA-2 Algoritmo Hash Seguro 2. 8

SHA-256 Algoritmo Hash Seguro 256. 8

Capítulo 1

Introdução

A segurança cibernética se tornou um campo de extrema importância em nossa sociedade cada vez mais digitalizada, onde a proteção de dados confidenciais é fundamental. As senhas, como um dos principais mecanismos de segurança, desempenham um papel crucial na proteção contra acessos não autorizados e ataques cibernéticos. No entanto, o constante avanço do poder computacional e o desenvolvimento de algoritmos sofisticados de quebra de senhas representam desafios significativos para a segurança das informações.

O Hashcat [1], uma ferramenta de recuperação de senhas amplamente utilizada, consegue explorar o poder de processamento paralelo das GPUs para acelerar a quebra de *hashes* criptográficos. Essa abordagem permite testar um grande número de combinações de senhas em um curto período, tornando-se uma ferramenta valiosa para auditores de segurança e pesquisadores da área.

No entanto, a utilização eficiente do Hashcat em larga escala apresenta desafios, como a necessidade de gerenciar múltiplas GPUs, otimizar o uso de recursos computacionais e garantir a segurança e a privacidade dos dados envolvidos no processo de quebra de senhas [2]. A complexidade de configuração e gerenciamento de ambientes distribuídos para a execução do Hashcat pode ser um obstáculo para muitos usuários.

Este trabalho propõe o desenvolvimento de uma plataforma inovadora, denominada Hashcat como Serviço (HaaS), que visa simplificar e otimizar o processo de quebra de senhas, tornando-o mais acessível e eficiente. A plataforma HaaS combina o Hashcat com

tecnologias de *containers* e o *middleware* rOpenCL, para criar um serviço escalável, de alto desempenho e fácil utilização. Através da interface web, os usuários podem iniciar e gerenciar instâncias do Hashcat em *containers*, aproveitando o poder de processamento de GPUs locais e remotas de forma eficiente.

A utilização de *containers* permite o isolamento e a portabilidade das instâncias do Hashcat, garantindo a segurança e a flexibilidade do sistema. O rOpenCL facilita o gerenciamento e a utilização de recursos de hardware heterogêneos, como GPUs de diferentes fabricantes e modelos, locais e remotas, maximizando o desempenho do sistema.

Com a plataforma HaaS, espera-se democratizar o acesso a ferramentas de recuperação de senhas de alto desempenho, permitindo que pesquisadores, auditores de segurança e profissionais da área possam realizar análises e testes de forma mais eficiente e segura. Além disso, a plataforma oferece recursos avançados, como o escalonamento automático de recursos e a migração de instâncias entre nós de computação, para garantir a alta disponibilidade e o desempenho otimizado do serviço.

1.1 Contribuições

Esta dissertação contribui para o campo da segurança de senhas e computação de alto desempenho ao apresentar o HaaS, um serviço inovador que aproveita o poder da computação heterogênea e distribuída para otimizar a recuperação de senhas. O HaaS é uma plataforma que simplifica e acelera o processo de recuperação de senhas, tornando-o acessível a usuários não técnicos.

A implementação do HaaS como um sistema distribuído permite utilização de múltiplas máquinas e dispositivos heterogêneos, maximiza o poder computacional e reduz o tempo de recuperação de senhas. Adicionalmente, o HaaS permite o lançamento do Hashcat sob demanda, através da utilização de *containers*, com a possibilidade de migração entre os nós computacionais diferentes, impactando positivamente o desempenho do sistema e a recuperação de senhas. Além disso, a criação de uma interface gráfica intuitiva facilita a interação do usuário com o sistema, permitindo o gerenciamento de dispositivos,

execução de tarefas e monitoramento do progresso.

A incorporação de mecanismos de reserva garante a continuidade das operações, aumentando a confiabilidade do serviço. Por fim, a avaliação abrangente, com testes e *benchmarks*, demonstra a eficácia do HaaS em diferentes cenários, comprovando sua capacidade de recuperar senhas de forma eficiente e escalável.

Em resumo, esta dissertação não apenas apresenta uma solução prática para um problema relevante, mas também expande o conhecimento na área ao combinar tecnologias existentes de maneira inovadora e eficiente. O HaaS tem o potencial de se tornar uma ferramenta valiosa para profissionais de segurança e usuários comuns que precisam recuperar senhas perdidas ou comprometidas.

1.2 Relações com Trabalhos Anteriores

Esta dissertação tira partido do rOpenCL [3], uma solução que permite a uma aplicação heterogênea pré-compilada, assente em OpenCL, acesso transparente a co-processadores remotos, expostos de forma individualizada. Neste trabalho, o rOpenCL permitiu compartilhamento de dispositivos locais e remotos, e facilitou a migração de *containers*, permitindo manter uma visão consistente sobre o conjunto de GPUs usados.

1.3 Metodologia de Trabalho

A metodologia de trabalho proposta assentou em cinco etapas essenciais, cada uma delas contribuindo para a consecução bem-sucedida do projeto. São elas:

1. **Estudo e Experimentação com Hashcat e GPUs**: A primeira etapa desta metodologia focou-se na compreensão aprofundada do Hashcat, e na sua utilização com GPUs. Durante esta fase, validaram-se também os mecanismos de *checkpointing* do Hashcat, essenciais para suportar a mudança do conjunto de GPUs usados, bem como a migração de instâncias do Hashcat para outros nós computacionais.

2. **Desenvolvimento do Serviço Web do HaaS**: Ultrapassada a primeira etapa, avançou-se para o desenvolvimento de um serviço web, projetado para demonstrar a aplicação prática das técnicas aprendidas durante a experimentação com Hashcat e GPUs. Durante esta fase, foram elaborados os requisitos, a arquitetura e a implementação do serviço, com ênfase na segurança e na eficiência.

3. **Adição de Isolamento por Meio de *Containers***: Para garantir a escalabilidade, portabilidade e isolamento dos componentes do serviço web, esta etapa envolveu a utilização de tecnologias de *containers*. Estes oferecem um ambiente controlado e independente para a execução de aplicativos, permitindo a fácil implantação e gerenciamento dos componentes do serviço web. Isso também contribui para a segurança e a confiabilidade do sistema.

4. **Integração do rOpenCL com os *containers***: Por fim, a metodologia culmina na integração harmoniosa do rOpenCL com os *containers*. Esta integração é fundamental para garantir que o serviço tire total proveito das capacidades de processamento paralelo fornecidas pelo ambiente distribuído (acesso a GPUs locais e remotas) enquanto mantém a escalabilidade e a flexibilidade dos *containers*.

Esta metodologia, abrangente e progressiva, proporcionou um caminho claro para o sucesso na implementação do serviço web, que utiliza tecnologias de ponta, como Hashcat, GPUs, Docker e rOpenCL. Cada etapa contribuiu para a criação de um serviço robusto, seguro e eficiente, capaz de atender às demandas de computação de alto desempenho.

1.4 Estrutura do Documento

O resto do documento está estruturado da seguinte forma:

- **Capítulo 2**: fornece o contexto conceptual e tecnológico do trabalho, explorando conceitos nucleares, trabalho relacionado, e abordando as ferramentas mais relevantes para o desenvolvimento do projeto;

- **Capítulo 3**: descreve, em detalhe, os aspetos mais importantes da implementação, abrangendo desde experimento iniciais destinados a aferir a viabilidade do pretendido, até à implementação do serviço web e sua infraestrutura, mecanismos de reserva e interface gráfica;
- **Capítulo 4**: é dedicado aos testes e à avaliação do sistema, detalhando a metodologia, o ambiente de testes e os resultados obtidos;
- **Capítulo 5**: conclui a dissertação, recapitulando as descobertas, refletindo sobre as lições aprendidas e sugerindo possíveis direções para trabalhos futuros;
- **Apêndice A**: contém a proposta original do trabalho;
- **Apêndice B**: contém o manual de instalação e utilização do HaaS, em inglês;
- **Apêndice C**: contém os ficheiros Dockerfile mais relevantes;
- **Apêndice D**: contém os ficheiros Docker Compose mais relevantes;
- **Apêndice E**: contém um exemplo dos *scripts* usados no *benchmarking*.

Capítulo 2

Conceitos e Tecnologias

Compreender o contexto desta pesquisa, e as ferramentas utilizadas, é fundamental para apreciar o valor e a originalidade das suas contribuições. Nesse sentido, este capítulo debruça-se sobre os principais conceitos em torno dos quais gira a dissertação, e passa em revista as principais tecnologias exploradas, de cuja combinação resultou o produto final.

2.1 Senhas e *Hashes*

Senhas são combinações de caracteres, como letras, números e símbolos, que são usadas para autenticar a identidade de um utilizador e conceder acesso a sistemas, contas pessoais, aplicativos, dispositivos e recursos online. Elas desempenham um papel fundamental na segurança cibernética e na proteção das informações pessoais e sensíveis, sendo portanto críticas na autenticação e na proteção de informações confidenciais.

A força de uma senha é determinada pelo seu comprimento e complexidade. Senhas curtas e compostas apenas por letras minúsculas podem ser quebradas rapidamente por ataques de força bruta [4]. É por isso que é crucial incentivar o uso de senhas alfanuméricas, que incluam letras maiúsculas, números e caracteres especiais. Essa complexidade adicional aumenta exponencialmente o número de combinações possíveis e, consequentemente, o tempo necessário para um ataque de força bruta ter sucesso.

Armazenar senhas em texto claro é arriscado: uma violação de segurança que permita

aceder ao ficheiro com as senhas irá comprometê-las. Para mitigar esse risco, as senhas são transformadas em *hashes* criptográficos antes de serem armazenadas. *Hashes* [5] são valores resultantes da aplicação de um algoritmo criptográfico a um conjunto de dados, como uma senha, a fim de os representar de forma compacta única e irreversível.

2.1.1 Algoritmos de *Hashing*

Os algoritmos de *hashing* são algoritmos criptográficos que transformam os dados originais em uma sequência de caracteres alfanuméricos de comprimento fixo, que parece aleatória – um *hash*. Isso adiciona uma camada extra de segurança, pois caso um atacante obtenha acesso ao repositório com os *hashes*, não terá acesso direto às senhas.

Um dos algoritmos de *hashing* mais antigos e utilizados é o Algoritmo de Resumo de Mensagens 5 (MD5) [6]. Foi, e ainda é, bastante utilizado para criptografia de senhas. Porém, hoje em dia é considerado um algoritmo relativamente fraco, pois é vulnerável a colisões (situação em que duas entradas diferentes geram o mesmo resultado).

Outros algoritmos de *hashing* bastante utilizados são os algoritmos da família Algoritmo Hash Seguro 2 (SHA-2), como o Algoritmo Hash Seguro 256 (SHA-256) [7] por exemplo. Este é um algoritmo considerado seguro, por não ser vulnerável a colisões.

Porém, tanto o SHA-256 quanto o MD5 não são algoritmos considerados seguros para armazenamento de senhas (apesar de o SHA-256 ser considerado seguro para outros cenários de uso). Isso porque existem algoritmos mais eficazes e indicados para estes propósitos, como o BCrypt [8], Scrypt [9] e o Argon2 [10]. Esses algoritmos são projetados especificamente para o armazenamento seguro de senhas e são mais resistentes a ataques de força bruta, funcionando de maneira mais sofisticada, incorporando práticas como o uso de *salts* (processo em que uma *string* aleatória é adicionada à *string* original a fim de dificultar a quebra de senha), para proteger as senhas de maneira mais eficaz.

2.1.2 Ameaças às Senhas

Dado o seu relevante papel ao autenticar e proteger as informações de utilizadores, as senhas são de grande interesse para cyber-criminosos (e, frequentemente, alvo de ataques cibernéticos), sendo a sua proteção um dos maiores desafios de segurança enfrentados por organizações e indivíduos. Algumas ameaças comuns às senhas são:

1. **Ataques de Força Bruta e Ataques de Dicionário:** Ataques de força bruta [11] envolvem a geração de todas as combinações possíveis de caracteres para adivinhar uma senha; dependendo da complexidade da senha original, podem ser impraticáveis em tempo útil. Os ataques de dicionário [12] usam diretamente palavras reais ou sequências comummente usados como senhas, sendo mais rápidos.

2. **Ataques de *Rainbow Table*:** Estes ataques envolvem o uso de tabelas pré-construídas (*Rainbow Tables*) [13] que mapeiam *hashes* de senhas conhecidas para as próprias senhas. Isso permite a recuperação muito rápida de senhas a partir de *hashes*, desde que se possua o *hash* da senha que se quer recuperar para comparar.

3. **Engenharia Social:** Os ataques de engenharia social [14] exploram a manipulação psicológica de indivíduos para obter as suas senhas. Isso pode incluir *phishing*, pelo qual os atores mal intencionados enganam os utilizadores para revelar suas credenciais.

4. **Vazamentos de Dados:** Quando informações de *login* são vazadas ou roubadas devido a brechas de segurança [15], elas podem ser usadas em ataques subsequentes em outras contas, devido à reutilização de senhas (prática, infelizmente, comum).

2.2 Quebra Legítima de Senhas

As técnicas criptográficas, e senhas associadas, desempenham um papel fundamental na proteção de dados e na segurança das comunicações *online*. No entanto, em certos casos,

é necessário ou desejável quebrar a criptografia / senhas para atingir objetivos legítimos de segurança cibernética. Situações em que essa quebra é legítima ou aceitável incluem:

- **Investigação de crimes cibernéticos**: As autoridades policiais e agências de segurança nacional podem precisar quebrar a criptografia / recuperar senhas para acessar informações vitais em investigações criminais, como fraudes, tráfico de drogas, terrorismo e outras atividades ilegais.

- **Testes de penetração e avaliações de vulnerabilidades**: Empresas e organizações contratam especialistas em segurança cibernética para avaliar a segurança de seus sistemas. Isso pode envolver a tentativa de quebrar a criptografia para identificar vulnerabilidades e corrigi-las antes que sejam exploradas por invasores maliciosos.

- **Recuperação de dados perdidos**: Em casos de perda de dados devido a falhas em dispositivos de armazenamento ou senhas esquecidas, ferramentas de recuperação de dados podem ser usadas para recuperar informações criptografadas.

- **Pesquisa acadêmica e desenvolvimento de normas**: A pesquisa na área de criptografia frequentemente envolve a análise e quebra de algoritmos para melhorar a segurança ou desenvolver novas normas criptográficas.

2.2.1 Algoritmos e Ferramentas

A quebra de senhas, cada vez mais facilitada por via de novas e mais sofisticadas técnicas, é uma ameaça significativa à segurança da informação, pois o acesso não autorizado a sistemas e dados confidenciais pode levar a perdas financeiras, violação de privacidade, roubo de identidade e comprometimento da reputação de empresas e indivíduos.

Com o avanço contínuo do poder computacional e o desenvolvimento de novos algoritmos e ferramentas de quebra de senhas cada vez mais sofisticados, a eficácia das medidas tradicionais de segurança baseadas em senhas está sendo desafiada.

Alguns avanços nos algoritmos de quebra têm origem em estudos já realizados há algum tempo, como é o caso do Modelo de Markov, proposto em 2005 [16], que tem como objetivo encontrar o próximo caracter de uma senha usando probabilidades.

Outra técnica, proposta em 2009 [17], inclui o uso de um modelo baseado em Gramática Livre de Contexto e Probabilística para gerar regras de manipulação que ajudem a alcançar a senha mais rapidamente.

Mais recentemente, recorreu-se ao uso de Inteligência Artificial, com o aproveitamento de Rede Generativa Adversária [18], e ao uso da similaridade de fragmentos para melhorar o desempenho do modelo baseado em Gramática Livre de Contexto e Probabilística [19].

Os algoritmos e técnicas de quebra encontram expressão em muitas ferramentas, disponíveis quer para atores mal-intencionados, quer para os auditores de segurança legítimos. Algumas dessas ferramentas elencam-se a seguir:

- **Cain & Abel** [20]: ferramenta de quebra de criptografia bastante conhecida, com a sua última versão lançada já em 2014 [21]; distribuída só para Windows, é capaz de realizar ataques de força bruta, de dicionário ou com base em *Rainbow Tables*;

- **RainbowCrack** [22]: ferramenta focada no uso de *Rainbow Tables* para quebra de senhas; inicialmente disponível apenas para Windows, em 2014 foi desenvolvida uma implementação para o Kali Linux [22]; seu uso é voltado para a quebra de senhas mais simples; Yuki Tabata e outros [23], propuseram uma extensão do RainbowCrack para tirar proveito também do uso de GPUs;

- **Ophcrack** [24]: outra alternativa focada apenas na quebra de senhas de contas Windows e usando também *Rainbow Tables*;

- **Aircrack-ng**[25]: ferramenta que se concentra na quebra de senhas de redes WiFi;

- **THC Hydra** [26]: disponível para Windows e Linux, e desenvolvida para atuar de forma *online*, esta ferramenta suporta uma vasta quantidade de protocolos e pode realizar ataques de dicionário e força bruta; é uma ferramenta amplamente utilizada para realizar quebra de senhas *online*;

- **NCrack** [27]: tal como a ferramenta THC Hydra, foi também desenvolvida para atuar de forma *online* e é multiplataforma; possui suporte a uma variedade de protocolos, incluindo HTTP, SSH, RDF, FTP, MQTT e outros;

- **John the Ripper** [28]: ferramenta muito popular para o uso *offline*, de código aberto, multiplataforma, bastante flexível e com Interface Gráfica do Utilizador (GUI) disponível; oferece suporte a diversos tipos de algoritmos criptográficos, e conta com uma distribuição para a nuvem através de uma *template* (imagem de máquina virtual pré-gerada) para instanciação na plataforma comercial AWS [29];

- **Hashcat** [1]: amplamente utilizada para quebra de senhas, é auto-proclamada como a ferramenta desse tipo mais rápida do mundo e a primeira com um mecanismo de regras executado em *kernels* de GPUs, com suporte a GPUs de diferentes fabricantes, proporcionando um desempenho otimizado e recursos exclusivos para quebrar senhas de forma eficiente; de código aberto e sendo utilizada sob a licença MIT [30], também é multiplataforma e possui suporte a mais de 350 algoritmos criptográficos.

Entre as ferramentas mencionadas, John the Ripper e Hashcat destacam-se como estado da arte, sendo amplamente utilizadas e frequentemente atualizadas, com correções de erros e novas funcionalidades. Existem, todavia, diferenças significativas entre as duas.

A aplicação John the Ripper é flexível, oferecendo configurações personalizadas e uma interface gráfica amigável para utilizadores que "fogem" da linha de comando. No entanto, carece de funcionalidades mais avançadas, como as que são oferecidas pelo Hashcat, e que ditaram a escolha deste como a plataforma de quebra de senhas adotada nesta dissertação.

Uma das principais características diferenciadores do Hashcat é a capacidade de utilizar o poder de processamento de GPUs, acelerando significativamente o processo de quebra de senhas. Esta capacidade é particularmente útil para aumentar a eficiência e reduzir o tempo necessário para recuperar senhas complexas. Neste contexto, o Hashcat é compatível com *backends* CUDA [31] e OpenCL [32]. O primeiro permite explorar GPUs da NVIDIA de forma otimizada. O segundo permite explorar quaisquer aceleradores expostos pelo *backend* OpenCL (não só GPUs, incluindo de fabricantes que não a

NVIDIA, como Unidades Centrais de Processamento (CPUs) *multicore* e dispositivos de tipo FPGA[1], isolados ou combinados entre si).

A arquitetura do Hashcat é bastante eficiente e flexível, suportando diferentes modos de ataque, incluindo de força bruta, de dicionário, a utilização de regras customizadas e a combinação de diferentes modos. Esta variedade de modos permite adaptar as estratégias de ataque de acordo com as especificidades dos *hashes* e as condições do cenário, otimizando o espaço de busca e aumentando a eficiência do processo. No modo de ataque de dicionário, por exemplo, o Hashcat pode aplicar regras complexas de manipulação de palavras para gerar variações de senhas, enquanto o modo de força bruta explora todas as combinações possíveis de caracteres.

A interface de linha de comando do Hashcat permite controle granular dos parâmetros de ataque, suportando a personalização detalhada das estratégias de recuperação. Funcionalidades como "pause", "restore" e "checkpoint" facilitam o gererenciamento de ataques de longa duração, garantindo a continuidade e a recuperação do progresso em caso de interrupções. O uso de arquivos de apoio (*.restore* para retomar ataques interrompidos e *.potfile* para registrar senhas já encontradas), otimiza o tempo e o uso dos recursos computacionais.

Adicionalmente, o Hashcat possui uma estrutura de arquivos bem definida: o arquivo de entrada contém os *hashes* a serem quebrados, enquanto o arquivo de saída armazena as senhas recuperadas com sucesso. O Hashcat também permite a definição de máscaras personalizadas, que refinam a geração de senhas candidatas e otimizam o espaço de busca, aumentando a eficiência do processo.

Estas características fazem do Hashcat a escolha ideal para este trabalho. O suporte a uma ampla gama de algoritmos criptográficos e o desempenho superior são fatores cruciais para a quebra de senhas como serviço. O Hashcat oferece uma plataforma robusta e eficiente para testar a resistência de senhas contra ataques cibernéticos, proporcionando maior liberdade de escolha e otimização dos recursos computacionais para os utilizadores.

[1]Field Programmable Gate Array - ver `https://en.wikipedia.org/wiki/Field-programmable_gate_array`

Isso possibilita uma análise aprofundada das vulnerabilidades e o desenvolvimento de estratégias mais eficazes para mitigar riscos e fortalecer a segurança cibernética.

2.2.2 Sistemas Heterogêneos para Quebra de Senhas

Sistemas computacionais heterogêneos integram componentes, elementos ou partes que são diferentes em natureza, tipo, arquitetura, ou que operam de maneira distinta. Isto em contraste com sistemas homogêneos, nos quais os componentes são semelhantes e interagem de maneira uniforme [33]. A plataforma desenvolvida neste trabalho (HaaS) assume a sua instanciação em sistemas heterogêneos, que combinam Unidade Central de Processamentos (CPUs) com GPUs [33].

O uso de sistemas heterogêneos com GPUs para quebra de senhas é vantajoso pois as GPUs são especialmente eficientes nesse tipo de tarefa, devido à sua capacidade de processamento massivamente paralelo. Enquanto as CPUs clássicas são adequadas para tarefas gerais, as GPUs são capazes de realizar milhões de tentativas por segundo, tornando-as ideais para acelerar ataques de quebra de senha [34].

Trabalhos como GPU-based Password Cracking [35] evidenciam como o uso de GPUs permite alcançar desempenho muito superior ao uso exclusivo de CPUs. Ou ainda outros [36], que mostram como a ferramenta John the Ripper no modo ataque de dicionário pode ser significativamente mais rápido recorrendo ao uso de GPUs. Estes exemplos evidenciam como a exploração de sistemas heterogêneos pode acelerar a quebra de senhas.

Por isso, as ferramentas modernas, como o Hashcat, possuem a capacidade de utilizar o poder de processamento de várias GPUs, CPUs e FPGAs, em paralelo, para realizar ataques de força bruta, ataques de dicionário e outros tipo de ataques.

Standards de Exploração de Sistemas Heterogêneos

OpenCL [32] e CUDA [31] são dois padrões de exploração de sistemas heterogêneos, usados especialmente no contexto de computação de alto desempenho. Ambos permitem

aproveitar a potência de processamento de dispositivos *many-core* de tipo GPU, em cooperação com CPUs, para acelerar tarefas computacionais intensivas. No entanto, têm origens diferentes, e diferenças importantes. Assim, o CUDA é um padrão e ecossistema de computação heterogênea proprietário, da NVIDIA [37], direcionado ao uso das suas GPUs, sendo altamente eficiente para este propósito, e a norma *de facto*. Já o OpenCL é um padrão aberto e multiplataforma, originário do Khronos Group [38], que suporta uma ampla variedade de dispositivos (GPUs, CPUs e FPGAs e outros aceleradores), tendo ganho menos tração que o CUDA, apesar das vantagens evidentes que oferecia sobre este.

Acesso a Aceleradores Remotos via rOpenCL

Estudos como [39] apontaram, há algum tempo, para o interesse em possibilitar a execução remota de aplicações heterogêneas. Isto traz diversas vantagens para as aplicações, que passam a poder aproveitar recursos computacionais remotos, sendo especialmente útil quando o *hardware* local não é suficiente para as necessidades computacionais da aplicação.

Traz também potenciais benefícios em termos de uma maior escalabilidade e resiliência das aplicações. No entanto, é importante observar que a execução remota de aplicativos heterogêneos também apresenta desafios, como a latência da rede, a necessidade de gerenciar recursos compartilhados e questões de segurança.

Neste contexto emergiu o *remote OpenCL* [3], como uma solução que permite a exploração de aceleradores remotos por aplicações OpenCL pré-compiladas, com recurso a comunicação portável (TCP/IP). O próprio Hashcat foi já combinado com o rOpenCL [40], tendo-se demonstrado a capacidade deste em assegurar uma redução significativa do tempo de execução do Hashcat pela exposição e utilização de GPUs remotas. Além disso o rOpenCL cobre mais de 70% da funções do OpenCL, permitindo a este trabalho tirar proveito do rOpenCL sem a necessidade de alterações do código fonte.

A arquitetura do rOpenCL, mostrada na Figura 2.1, permite expor dispositivos remotos através do protocolo TCP/IP, e acesso aos mesmos através do *ICD Loader*, tornando desta forma o seu uso completamente transparente sob a perspetiva das aplicações clientes.

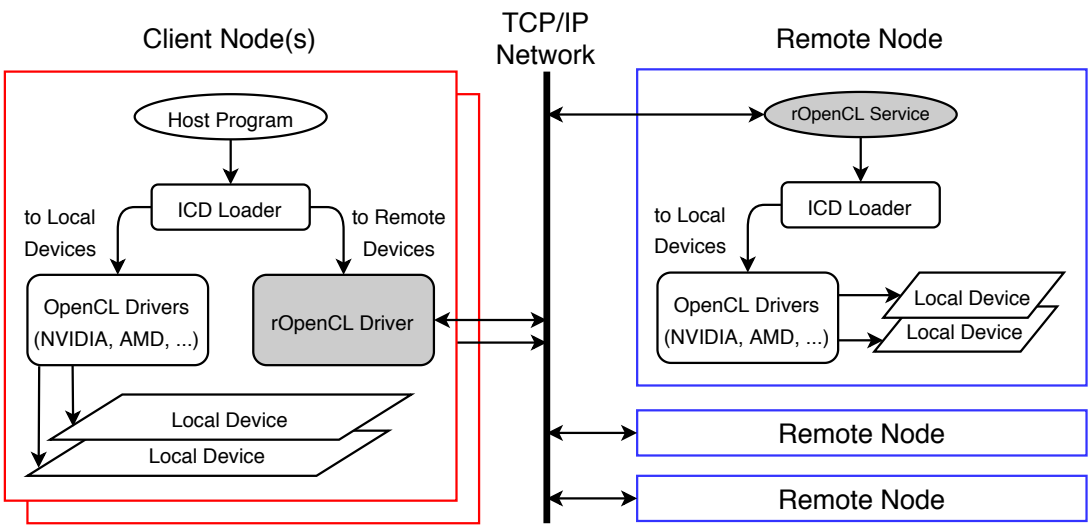


Figura 2.1: Arquitetura do rOpenCLL [40]

2.2.3 Sistemas Distribuídos para Quebra de Senhas

Sistemas distribuídos são sistemas de computação compostos por vários componentes autônomos interconectados, que trabalham juntos para realizar uma tarefa ou conjunto de tarefas. Esses componentes podem ser computadores individuais, servidores, dispositivos móveis ou quaisquer outros dispositivos de computação.

Vários trabalhos têm mostrado também benefícios do uso de sistemas distribuídos no contexto de quebra de senhas, pois permitem a exploração de uma vasta quantidade de combinações de senhas e *hashes* em paralelo, tornando os ataques ainda mais eficazes. Alguns exemplos desta abordagem começaram a surgir em 2009 com a utilização de servidores HPC com MPI para alcançar maior desempenho e escalabilidade [41], [42].

Estudos mais recentes exploram o uso de computação em nuvem. É o caso de [43], onde se apresenta uma prova de conceito para quebra de *hashes* baseada em nuvem.

Outra abordagem é apresentada em [44], onde se introduz um programa semelhante baseado em BOINC e um programa de autoria própria chamado Fitcrack, posteriormente alterado para usar o Hashcat [45]. O autor compara ainda o Fitcrack com o Hashtopolis [46], um sistema de gerenciamento de quebra de senhas distribuído com base no Hashcat, usado amplamente usado por profissionais de segurança cibernética.

O Hashtopolis é aliás considerado o estado da arte no contexto de quebra de senhas de forma distribuída, pois encapsula o Hashcat e promove funcionalidades adicionais, como por exemplo o gerenciamento distribuído, centralização de recursos, segurança, escalabilidade. Tem ainda uma grande comunidade ativa que o mantém em constante evolução.

O HaaS, desenvolvido neste trabalho, além da sua relação com os sistemas heterogêneos, é também enquadrável na categoria dos sistemas distribuídos. A grande diferença entre o HaaS e as ferramentas já apresentadas, como Hashtopolis e o Fitcrack, é que embora todas procurem aumentar o desempenho de cada execução por meio da distribuição, o HaaS permite que diversos utilizadores tirem proveito da aplicação, em simultâneo. De notar ainda que tais ferramentas não são diretamente comparáveis em termos de desempenho. Não são também mutuamente exclusivas, podendo ainda ser combinadas em prol de promover novas funcionalidades (estando isso fora do contexto deste trabalho).

2.3 Tecnologias Auxiliares

Nesta secção abordam-se as várias ferramentas e tecnologias que, adicionalmente ao Hashcat e ao rOpenCL, tornaram viável o desenvolvimento da plataforma HaaS.

2.3.1 Containerização

Uma tecnologia crucial que alavanca este trabalho é a containerização [47]. Trata-se de uma forma de virtualização a nível de sistema operacional, que permite isolar aplicativos e seus recursos, enquanto compartilham o *kernel* do sistema operacional subjacente.

Envolve o empacotamento de uma aplicação, junto com todas as suas dependências, bibliotecas e arquivos de configuração necessários para a sua execução em um ambiente auto-contido – o *container*. Isso garante que a aplicação funcione de maneira consistente em qualquer ambiente computacional (estações de desenvolvimento, servidores, máquinas virtuais), desde o desenvolvimento até a produção, facilitando a sua portabilidade.

No contexto deste trabalho, os *containers* são a ideais para criar instâncias de serviços de quebra de senhas, de acordo com a necessidade dos utilizador, de forma dinâmica. Além

disto, é possível definir, de forma simples, os recursos que estarão disponíveis dentro do *containers*, aos quais os utilizadores terão acesso.

Docker

O Docker [48] é uma das ferramentas de containerização mais populares. Embora existam alternativas, como o Podman [49], e Rkt [50], o Docker destaca-se devido a algumas vantagens, como: multiplataforma, facilidade de uso, ampla comunidade e ecossistema associados, padronização, níveis de segurança, facilidades de orquestração de *containers* via Docker Swarm [51] ou Kubernetes [52].

Por estes motivos, o Docker [48] é a plataforma de conteinerização utilizada no projeto. Cada componente do sistema, desde os microserviços responsáveis pela lógica de negócio até as ferramentas auxiliares, é encapsulado em seu próprio *container* Docker. Essa abordagem oferece uma série de benefícios cruciais:

- **Portabilidade**: os *containers* Docker são auto-suficientes, carregando consigo todas as dependências necessárias para a execução do software; isso garante que os componentes possam ser facilmente implantados em diferentes ambientes, sejam eles máquinas locais, servidores em nuvem ou *clusters* de computadores, sem a necessidade de configurações complexas e específicas para cada ambiente;

- **Isolamento**: cada *container* Docker opera num ambiente isolado, com seu próprio sistema de arquivos, rede e processos; esse isolamento garante que os componentes não interferem uns com os outros, evitando conflitos de dependências e problemas de compatibilidade: o isolamento contribui ainda para a segurança do sistema, limitando o impacto de possíveis vulnerabilidades ou ataques a um só *container* (salvo vulnerabilidades do próprio *container* que permitam ter acesso ao *host*);

- **Consistência**: o uso de *containers* Docker permite padronizar a forma de construção de cada *container*, mantendo as mesmas dependências (*frameworks*, linguagens de programação, bibliotecas, versões e outros); tal permite que o *container* seja criado em diferentes máquinas e locais, mantendo um funcionamento coerente.

A utilização do Docker assenta nalguns conceitos importantes, como:

- **Build**: O processo de criação de uma Imagem com base no Dockerfile criado (arquivo que contém as instruções de configuração do *container*);
- **Imagem**: O artefato gerado após o processo de *Build* ser realizado, e que servirá de base para a criação dos *containers*; é, portantto, uma espécie de *template*;
- **Container**: Uma instância particular da Imagem.

O processo de *Build* e criação do *container* podem ser demasiados complexo, pois podem necessitar de informações externas (como variáveis de ambiente), criação de volumes para partilha de ficheiros, exposição de portas de rede, etc. p2ara simplificar estes processos, o Docker disponibiliza a facilidade Docker Compose [53], que permite a criação de *containers* através de um único comando, e com base nas configurações de criação de um arquivo com extensão *yml* (ver Apêndice D para um exemplo usado neste trabalho).

Docker Swarm

O Docker ainda permite a utilização do Docker Swarm, uma ferramenta de orquestração de *containers*. Esta ferramenta permite a criação de um *cluster*, ligando diferentes máquinas e proporcionando a criação e gerenciamento de *containers*. O Docker Swarm é fundamental para o HaaS, pois é através dele que é realizada a criação e gerenciamento de *containers* de acordo com a demanda dos utilizadores.

De notar que o Docker Swarm não é a única opção de orquestração de *containers* disponível. O Kubernetes [52], já citado anteriormente, é umas das ferramentas mais utilizadas para esse fim. No entanto, o Kubernetes possui uma complexidade elevada, sendo ideal para arquiteturas complexas, as quais necessitam de funcionalidades de rede avançadas, configurações manuais do balanceador de carga, suporte a diferentes plataformas de containerização, entre outras facilidades, que não são relevantes para este trabalho.

O Docker Swarm possui assim um funcionamento simples, porém extensivo, que permite um desenvolvimento mais ágil, dado que não necessita de grande complexidade ao

configurar a plataforma. Outro fator que concorre para a adoção do Docker Swarm é a sua excelente interoperabilidade com o Docker, a qual é ampliada pela utilização da linguagem de programação go, comum a ambas as plataformas e ao HaaS.

2.3.2 Partilha de Ficheiros em Rede

Tipicamente, os *containers* necessitam de volumes para partilha de arquivos com o seu *host*, permitindo que os dados dos *containers* continuem armazenadas após o seu término. Esses volumes são simplesmente diretórios do *host*, que podem ser mapeados dentro do *container*. Porém, para que os dados não fiquem confinados a uma só máquina, este trabalho utiliza partilhas NFS [54], permitindo que os dados seja acessíveis via rede.

O uso do NFS para partilha de arquivos entre os diferentes nós do *cluster*, ajuda a garantir uma operação coesa e integrada do sistema distribuído. A utilização do NFS é essencial para que o Hashcat possa acessar os arquivos de entrada, como *hashes* a serem quebrados, dicionários, regras de ataque e outros dados necessários para a quebra de senhas, independentemente do nó em que esteja sendo executado.

O NFS assegura que todos os nós do cluster têm acesso consistente e atualizado aos arquivos compartilhados. Isso facilita a migração de tarefas entre nós, permitindo que um nó possa assumir a tarefa de outro em caso de falha, sem perda de progresso ou necessidade de reinicialização completa do processo. Esse mecanismo de compartilhamento de arquivos é particularmente importante para o funcionamento dos mecanismos de *checkpoint* e restauração do Hashcat, pois garante que as informações de estado e progresso estejam sempre disponíveis para qualquer nó que precise retomar uma tarefa.

Além disso, o NFS simplifica o gerenciamento de dados, centralizando os arquivos necessários para a operação do Hashcat e evitando a redundância e a necessidade de sincronização manual entre diferentes nós. Os resultados da quebra de senhas também são armazenados em um local centralizado, acessível a todos os nós do *cluster*, permitindo uma fácil consulta e análise posterior. A utilização do NFS no HaaS como sistema de arquivos centralizado, permite ainda adicionar novos nós ao *cluster* sem a necessidade de

re-compilar o HaaS para cada novo nó. Isso torna o sistema mais flexível e adaptável a mudanças na carga de trabalho e no número de recursos disponíveis.

Em resumo, a integração do NFS no HaaS é fundamental para garantir a consistência, disponibilidade e eficiência do sistema distribuído, permitindo que o Hashcat opere de maneira otimizada em um ambiente de *cluster*.

2.3.3 Bases de Dados

Outra forma de armazenamento de dados são as plataformas MongoDB [55] e Redis [56].

O MongoDB, um banco de dados NoSQL, é escalável e flexível, sendo ideal para lidar com crescimento e demandas variáveis.

O Redis é um banco de dados em memória de alta performance. Desempenha um papel fundamental no gerenciamento de informações em tempo real. A velocidade e a eficiência do Redis são essenciais para garantir uma rápida resposta do sistema e a otimização do uso dos recursos de *hardware*, especialmente em cenários de alta demanda.

A combinação do MongoDB e do Redis oferece uma infraestrutura de dados robusta e eficiente, capaz de lidar com as complexidades do processamento de sistemas distribuídos e garantir a alta disponibilidade e desempenho do sistema. A escolha do MongoDB permite um gerenciamento eficaz de grandes volumes de dados e suporte a consultas complexas, enquanto o Redis assegura um acesso rápido e eficiente a dados críticos em tempo real.

Essa combinação garante a flexibilidade necessária para lidar com diferentes requisitos de carga de trabalho e escalabilidade, ao mesmo tempo que mantém uma operação eficiente e confiável. Essa integração permite ainda uma experiência de utilizador otimizada, com tempo de resposta rápido e gerenciamento eficiente dos recursos disponíveis, atendendo às necessidades de um ambiente de computação distribuída de alta performance.

2.3.4 Serviço de Autenticação

Um dos requisitos da plataforma HaaS é um mecanismo de segurança, autenticando os utilizadores da aplicação, assim como garantindo que apenas o administrador do sistema

possa acessar determinadas funcionalidades.

Neste contexto, foi adotado o Keycloak como servidor de autenticação. O Keycloak [57] é uma solução de código aberto para Gestão de Identidade e Acesso (IAM) [58] que fornece recursos de autenticação, autorização e segurança para aplicativos e serviços. Desenvolvido pela Red Hat, é amplamente usado para adicionar recursos de gerência de identidade em aplicativos web e APIs.

Além de ser uma das ferramentas mais amplamente utilizadas no desenvolvimento de software, o Keycloak oferece maior segurança para guardar as informações dos utilizadores, pois além da aplicação de algoritmos de criptografia sofisticados, o armazenamento das informações é feito em contextos separados. Também oferece funcionalidades nativas para controle de acesso, agrupamento de utilizadores, restauração de credenciais, entre outros.

Dada a aposta da plataforma na segurança, optou-se por utilizá-la, junto a um serviço de encapsulamento desenvolvido internamente para abstrair a complexidade envolvida.

2.3.5 Facilidades de Comunicação

É necessário ainda que as diversas tecnologias utilizadas neste traballho operem de forma organizada e em conjunto. Para isso, é utilizado o estilo arquitetural Transferência de Estado Representacional (REST) [59], que define uma estrutura para o protocolo Protocolo de Transferência de Hipertexto (HTTP).

Uma outra tecnologia a utilizada é o API Gateway Kong, pois acrescenta ao projeto organização e escalabilidade. Em um cenário de micro serviços [60], como o utilizado neste projeto, o número de serviços internos comunicando entre si pode crescer demasiadamente, assim como o número de requisições para estes serviços internos. Para evitar que isto se torne um problema, o Kong [61] é uma escolha adequada para o gererenciamento de REST API, funcionando como um gestor das requisições recebidas, distribuindo cada requisição para o serviço correto. Além disso, o Kong também funciona como um distribuidor de carga, permitindo que o sistema possa escalar sem precisar de modificações.

O Kong ainda possui uma integração direta ao Keycloak, fazendo com que as validações

de credenciais aconteçam a nível de *middleware*. Ou seja, quando a requisição é feita, o utilizador de uma rota protegida será validado pelo Kong antes que essa requisição seja propagada ao serviço correto, gerando um erro de permissão caso o utilizador não tenha as permissões corretas.

2.3.6 Linguagens de Programação

Para interligar as ferramentas descritas nas secções anteriores e prover o sistema, a linguagem Go é utilizada no servidor, enquanto o TypeScript é a opção para o cliente.

A linguagem Go [62] (desenvolvida pela Google) foi escolhida para este trabalho porque é uma linguagem desenvolvida com a proposta de ser simples e altamente orientada à concorrência, providenciando mecanismos que criam e gerenciam *threads* de forma otimizada, e canais, que permitem a comunicação entre *threads* de forma tammbém otimizada. Tudo isso de forma simplificada, sem a necessidade de demasiadas configurações .

Além disso, o Go é uma linguagem compilada (com vantagens a nível de desempenho), sendo os passos necessários para o *build* do projeto relativamente simples, e possui gerenciador de memória dinâmico. Outro fator para a escolha do Go é o fato de a sua sintaxe ser similar ao C, o que diminui a curva de aprendizado.

O TypeScript [63], um *superset* de JavaScript [64], é uma das linguagens padrões para o desenvolvimento *front-end*, sendo usada pelas principais tecnologias atuais de desenvolvimento Web (Angular, Vue e React). Isso porque o JavaScript resultante da tradução do TypeScript pode ser utilizado nativamente em qualquer navegador web moderno. O TypeScript inclui ainda o suporte ao uso de tipagem explícita [65] (o JavaScript possui tipagem implícita), melhorando a legibilidade do código e facilita a deteção de erros.

Um fator para a ampla adoção do JavaScript, e logo também do TypeScript, é a manipulação do DOM, que permite a manipulação da estrutura da página web em tempo real. Isso possibilita realizar animações, validações e atualizações de apenas alguns componentes da página, sem a necessidade de atualizar e redesenhar novamente toda a página.

Capítulo 3

Desenvolvimento

Este capítulo foca o desenvolvimento do HaaS, detalhando a jornada, desde os experimentos iniciais com o Hashcat, até à implementação completa da infraestrutura do serviço.

A experimentação inicial, abrangendo máquinas físicas, virtuais, *containers* e diversos GPUs, permitiu compreender o comportamento e desempenho do Hashcat em diferentes ambientes, culminando na escolha de *containers* com *drivers* específicos da NVIDIA para Docker, como o ambiente mais apropriado para o desenvolver e instanciar o HaaS.

De seguida, o HaaS foi construído, abrangendo o gerenciamento de utilizadores, de GPUs e integração do Hashcat. Mecanismos de reserva foram incorporados para privilegiar o acesso dos *containers* às GPUs, e um *cluster* foi criado para distribuir o processamento e otimizar o desempenho. A configuração do Docker Swarm e a capacidade de migração de *containers* entre os nós do *cluster* foram essenciais para alcançar essa otimização. Implementaram-se mecanismos de comunicação eficientes entre os componentes, e desenvolveu-se uma interface gráfica intuitiva para tornar o HaaS acessível e fácil de usar.

A conclusão do capítulo destaca os principais resultados do desenvolvimento, preparando o terreno para os testes e avaliações que serão apresentados no capítulo seguinte.

3.1 Experimentos Iniciais

O passo inicial para o desenvolvimento deste trabalho foi a experimentação e investigação das ferramentas a serem utilizadas, bem como a sua integração a fim de trabalharem em sintonia. Em particular, compreender como funciona o mecanismo de *checkpoint* e restauro do Hashcat foi crucial, incluindo a sua compatibilidade com mudanças na composição do conjunto de GPUs usados após o início de uma quebra de senhas.

3.1.1 *Checkpoint* e Restauro de Estado no Hashcat

O Hashcat suporta mecanismos de *checkpoint* e restauro do estado da sua execução. O primeiro grava o estado atual do processo de quebra de senha(s) em curso, parando de seguida a execução. O segundo permite, com base no arquivo `.restore` produzido pelo primeiro, o restauro do estado e a retoma do processo de quebra a partir desse ponto.

Essas funcionalidades não dependem necessariamente uma da outra: é possível realizar o restauro sem que tenha existido um *checkpoint*, desde que haja um arquivo de restauro com a estrutura e conteúdo válidos. Porém, juntas, ambas as funcionalidades são cruciais para a mudança de GPUs durante a execução, um requisito central para o trabalho.

O arquivo de restauro do Hashcat é um arquivo binário (logo, de mais difícil manipulação), com uma estrutura rígida e bem definida, de que pode ser visto um exemplo na Figura 3.1. Para facilitar a manipulação deste arquivo, utilizou-se a ferramenta `analyze_hc_restore` [66], que converte o arquivo para um formato legível, permitindo a análise e, caso seja necessário, a modificação das informações de restauro.

3.1.2 *Checkpoint* e Restauro em Máquinas Físicas e Virtuais

A validação do funcionamento dos mecanismos de *checkpoint* e restauro do Hashcat foi inicialmente realizada em uma só máquina, física. Testes com diferentes configurações de GPUs comprovaram a eficácia da troca de GPUs durante a execução.

Porém, repetindo o mesmo processo entre duas máquinas diferentes (neste caso, máquinas virtuais), vieram à tona várias limitações de ordem técnica, que impossibilitam

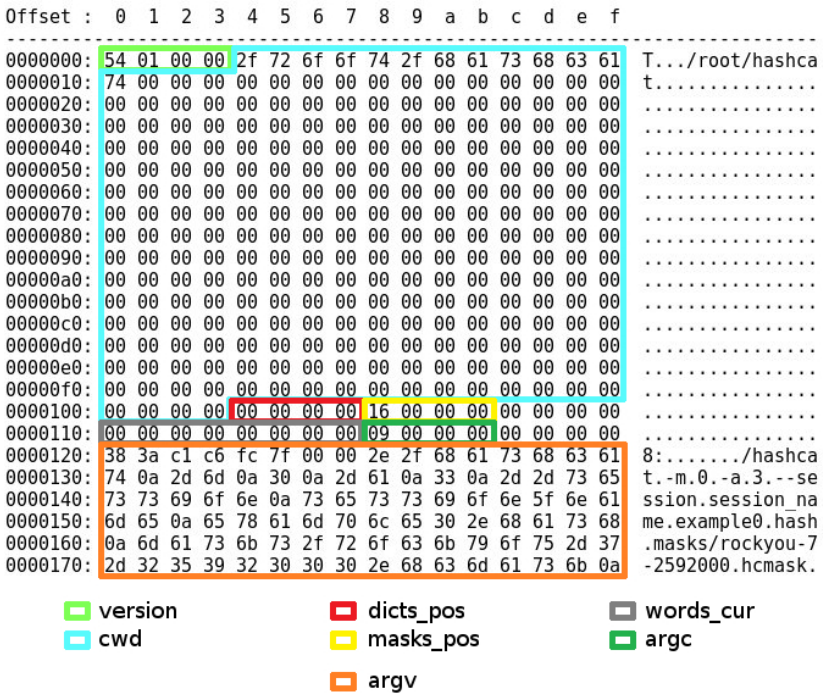


Figura 3.1: Arquivo de restauração.

um funcionamento transparente deste mecanismo. A primeira diz respeito ao acesso aos arquivos de restauro: para que se realize o restauro numa máquina diferente daquela onde foi realizado o *checkpoint*, os arquivos de restauro precisam ser copiados entre máquinas.

Contudo, não basta resolver o problema da cópia: é necessário garantir que ambas as máquinas tenham configurações do ambiente de execução (versões do Hashcat e bibliotecas associadas) semelhantes para um funcionamento correto do Hashcat. Além disso, também é necessário que os caminhos (no sistemas de ficheiros) embebidos no arquivo de restauro sejam válidos em ambas as máquinas; caso contrário, é preciso modificá-los após a cópia.

3.1.3 *Checkpoint* e Restauro em *Containers*

A experimentação com máquinas físicas e virtuais evidenciou a necessidade de garantir um ambiente de execução consistente. Para alcançar essa garantia, optou-se por adotar o *container* como mecanismo fornecedor do ambiente de execução do Hashcat.

Foram então realizados novos experimentos, envolvendo a criação de *containers* para

execução do Hashcat, fosse na mesma máquina, fosse em máquinas diferentes, e realizando o *checkpoint* num *container* e o restauro noutro. Comprovou-se a execução correta do Hashcat neste tipo de ambiente, ainda que a mobilidade do estado de execução (arquivos `.restore`) continuasse a ter de ser feita por cópia explícita entre *containers*.

De notar que não se optou pela replicação ou cópia integral do *container* original, mas sim pela criação de um novo *container* e cópia do estado do anterior. Esta opção, que tira partido da mobilidade do estado do Hashcat, e assume a destruição do *container* antigo e criação de um novo, é mais eficiente do que a cópia integral do *container* original. Isso porque ao realizar a cópia integral, além de copiar os arquivos já definidos, são copiados de ferramentas auxiliares e *logs*, que não afetam o comportamento do Hashcat; sendo assim, o volume de dados envolvidos numa cópia integral seria consideravelmente maior.

3.1.4 Acesso dos *Containers* às GPUs

A experimentação do Hashcat em ambiente de *container* necessitou de adaptações, pois tornou-se necessário o acesso do *container* às GPUs do *host*. Para tal, construiu-se um volume com os arquivos DRI (Drive Rendering Infrastructure) e KFD (Kernel Fusion Driver), adicionou-se o grupo *video* para garantir as permissões de utilização e dotou-se o *container* com os *drivers* necessários.

No entanto, quando está em causa o acesso a GPUs NVIDIA [37], tal é possível de forma mais simplificada, através de *drivers* específicos para containerização – NVIDIA Container Runtime [67]. Esta abordagem, além de simplificar a configuração dos *containers*, oferece maior portabilidade, sendo compatível não apenas com Docker, mas com uma variedade de outros provedores de serviços de containerização. Assim, durante estes experimentos iniciais, o acesso às GPUs NVIDIA através do NVIDIA Container Runtime, foi também uma abordagem explorada com sucesso.

Outra alternativa testada também nesta fase foi expor aos *containers* todas as GPUs do *cluster*, sejam locais ou remotas (instaladas noutros *hosts*), através do *middleware* rOpenCL. Este processo tornou mais eficiente o gerenciamento de GPUs, já que todas

passaram a ser vistos de forma global e transparente, sendo necessário apenas criar uma variável de ambiente com os endereços IP dos *hosts* onde as GPUs estavam alojadas.

3.1.5 Conclusões da Experimentação

Os experimentos iniciais permitiram comprovar um conjunto de pressupostos de carácter técnico que estavam na base da proposta deste trabalho: mobilidade de estado do Hashcat, compatibilidade com a sua containerização, possibilidade de integração com o rOpenCL.

A partir daqui, iniciou-se o projeto e desenvolvimento da plataforma HaaS, que permite a criação de instâncias do Hashcat em *containers* sob demanda dos utilizadores, utilizando o Docker para criação das instâncias e o rOpenCL para o uso global de GPUs.

3.2 Arquitetura do HaaS

O diagrama da Figura 3.2 exibe a arquitetura resultante da implementação descrita neste capítulo, e que está disponível em [68].

Os vários componentes assentam em tecnologias apresentadas no capítulo anterior. O papel destes componentes e as suas interações serão clarificadas nas secções seguintes.

De referir que os vários serviços que foi necessário desenvolver foram-no seguindo o estilo arquitetural de microserviços [60] (nesta abordagem, cada serviço é desenvolvido como um sistema individual, que se comunica com os outros através da rede). O resultado desta decisão arquitetural é uma plataforma com maior facilidade de manutenção.

Como já referido, no HaaS os *containers* Hashcat serão criados a pedido dos utilizadores. Para tal é necessário que também esteja a correr um *container* HaaS, que será o recetor dos pedidos dos utilizadores e mediador da comunicação entre estes e os *containers* Hashcat. É ainda necessário gerenciar as GPUs disponíveis, para que os *containers* Hashcat tirem proveito delas, independentemente da sua localização (visão Single-System Image) no *cluster*, o que se conseguiu com o *middleware* rOpenCL (ver Figura 3.3).

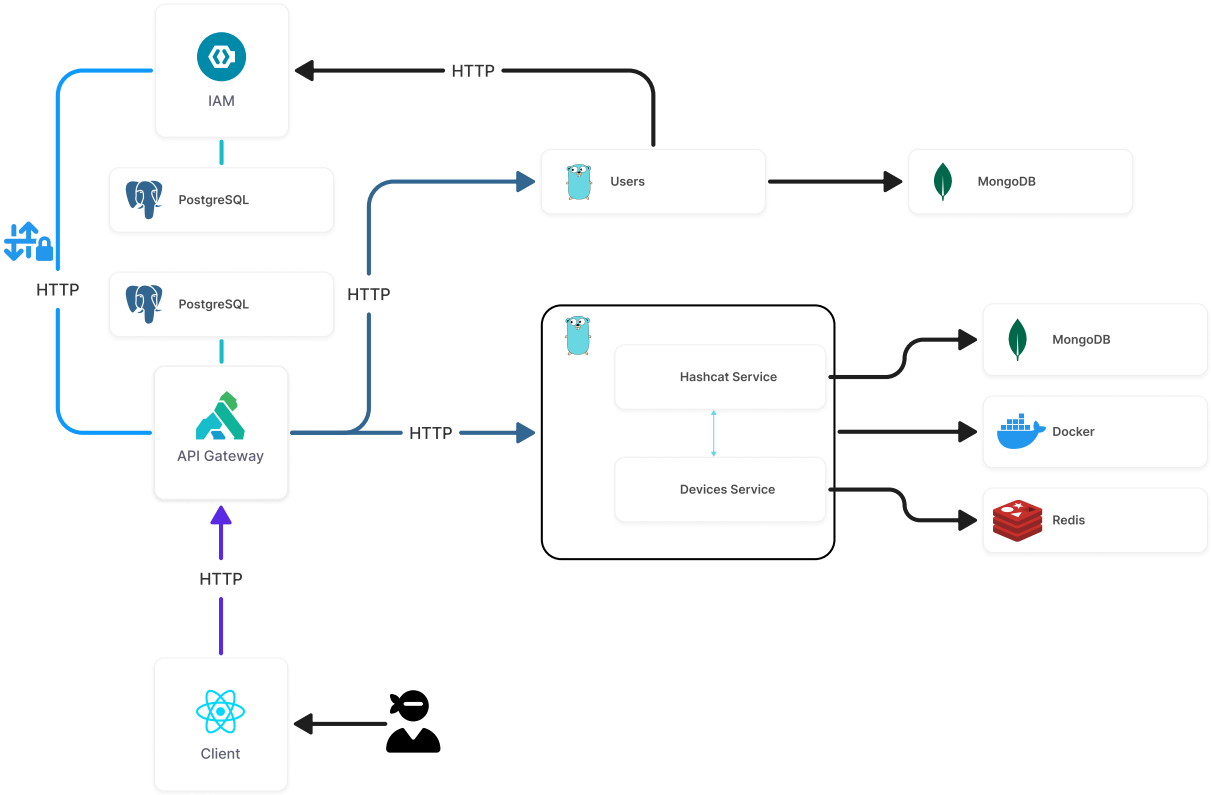


Figura 3.2: Diagrama de arquitetura.

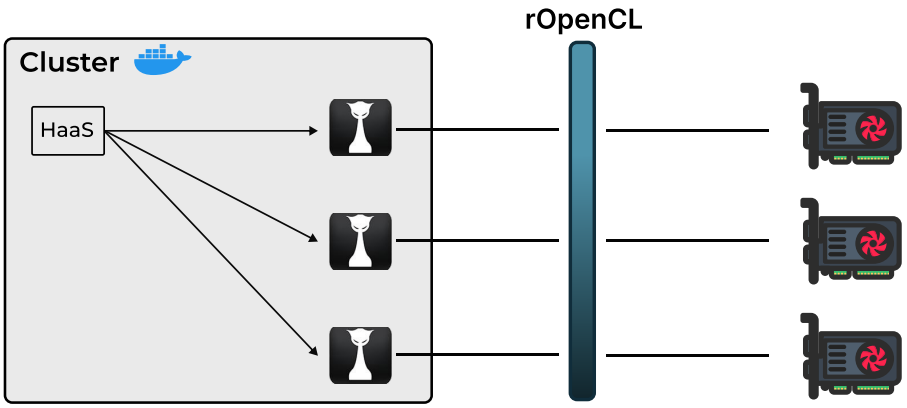


Figura 3.3: Single-System Image fornecida pelo rOpenCL.

3.3 Desenvolvimento do Backend

O primeiro passo no desenvolvimento da plataforma HaaS foi a criação da versão inicial necessária para a automação e orquestração dos processos. Este desenvolvimento visou

a implementação de uma REST API que suporte a execução distribuída do Hashcat em *containers*, garantindo eficiência, escalabilidade e portabilidade dos processos.

3.3.1 Infraestrutura de Suporte

A infraestrutura de suporte ao HaaS foi projetada para oferecer um ambiente robusto, escalável e seguro para a execução de tarefas de quebra de senhas em larga escala. A combinação de tecnologias de containerização, orquestração, gerenciamento de configuração e sistemas de arquivos distribuídos permite que o HaaS opere de forma eficiente, maximizando o uso de recursos e garantindo a alta disponibilidade do sistema. A Figura 3.4 mostra uma representação dessa infraestrutura, a uma escala mínima de dois nós físicos.

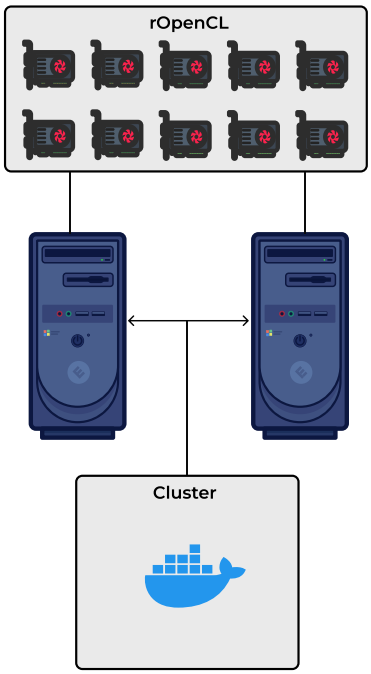


Figura 3.4: Diagrama de Infraestrutura.

A infraestrutura é instanciada com recurso a arquivos de tipo Docker Compose, juntamente com a implementação do HaaS, que podem ser vistos no seu repositório GITLAB [68]. Em cada arquivo de configuração estão definidas todas as dependências para o funcionamento de cada serviço. Além disso foi necessária a criação de um sub-rede de *overlay*

entre os serviços, para que possam comunicar entre si, uma vez que podem acabar por executar em *hosts* diferentes, por via de decisões tomadas pelo Docker Swarm.

Os arquivos de construção de cada microserviço constam do apêndice C, e as instruções de criação da infraestrutura do projeto através de arquivos *compose* estão no apêndice D.

3.3.2 Gerenciamento de Utilizadores

O gerenciamento de utilizadores compreende a sua autenticação e autorização. Para tal criou-se uma ligação entre a REST API, o Kong e o Keycloak (ver Figura 3.5).

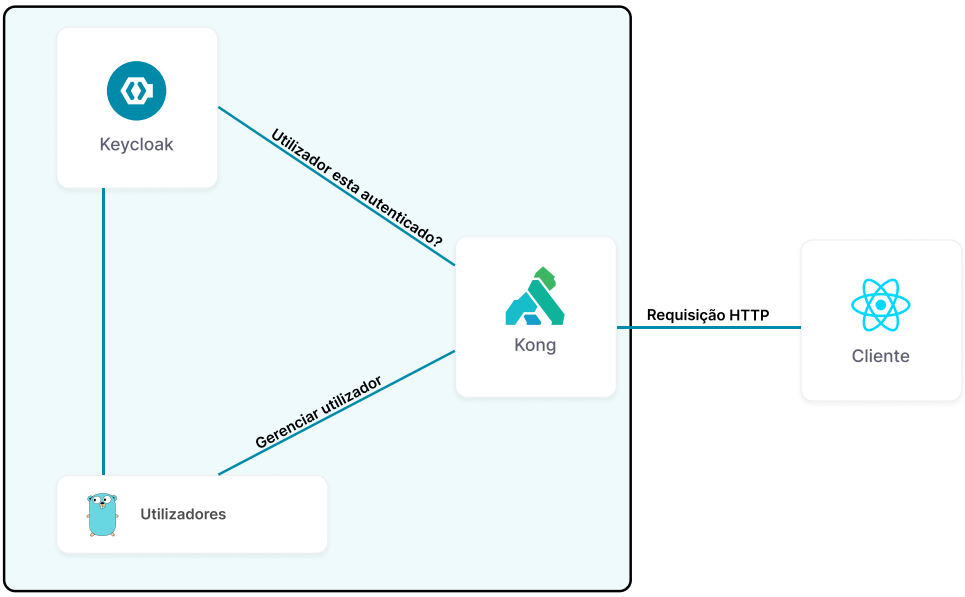


Figura 3.5: Gestão de utilizadores.

A autenticação e autorização do utilizador divide-se em três etapas. Na primeira, o utilizador cadastra-se na aplicação. Este processo é realizado através da REST API, que encapsula as funcionalidades do Keycloak, processando e redirecionando as requisições recebidas, criando o utilizador, e definindo as permissões e grupos de utilizadores.

Uma vez criado o utilizador, a segunda etapa é o acesso (*login*) ao sistema, o que envolve a validação das credenciais e receção de um *token* JWT (JSON Web Token). Este *token* será utilizado posteriormente, como forma indireta de validar as credenciais do utilizador, e autorizar o acesso aos vários componentes do sistema. O *token* JWT é

criado pelo Keycloak, onde também se define o seu tempo de vida. Uma vez expirado o *token*, é necessário realizar a validação das credenciais novamente.

Finalmente, na terceira etapa, o utilizador poderá tirar proveito da plataforma. Para isto, o acesso é validado com base no *token* JWT, processo realizado pelo Kong ao definir-se cada serviço disponível, que delimita o nível de acesso e integra-se ao Keycloak, validando o *token* recebido antes que a requisição possa ser redirecionada ao serviço correto.

Para que o serviço funcione corretamente, as rotas de criação e *login* não são protegidas: para estas rotas o acesso não tem nenhuma restrição quanto à identidade do utilizador.

O processo anteriormente descrito pode ser visto no diagrama da Figura 3.6.

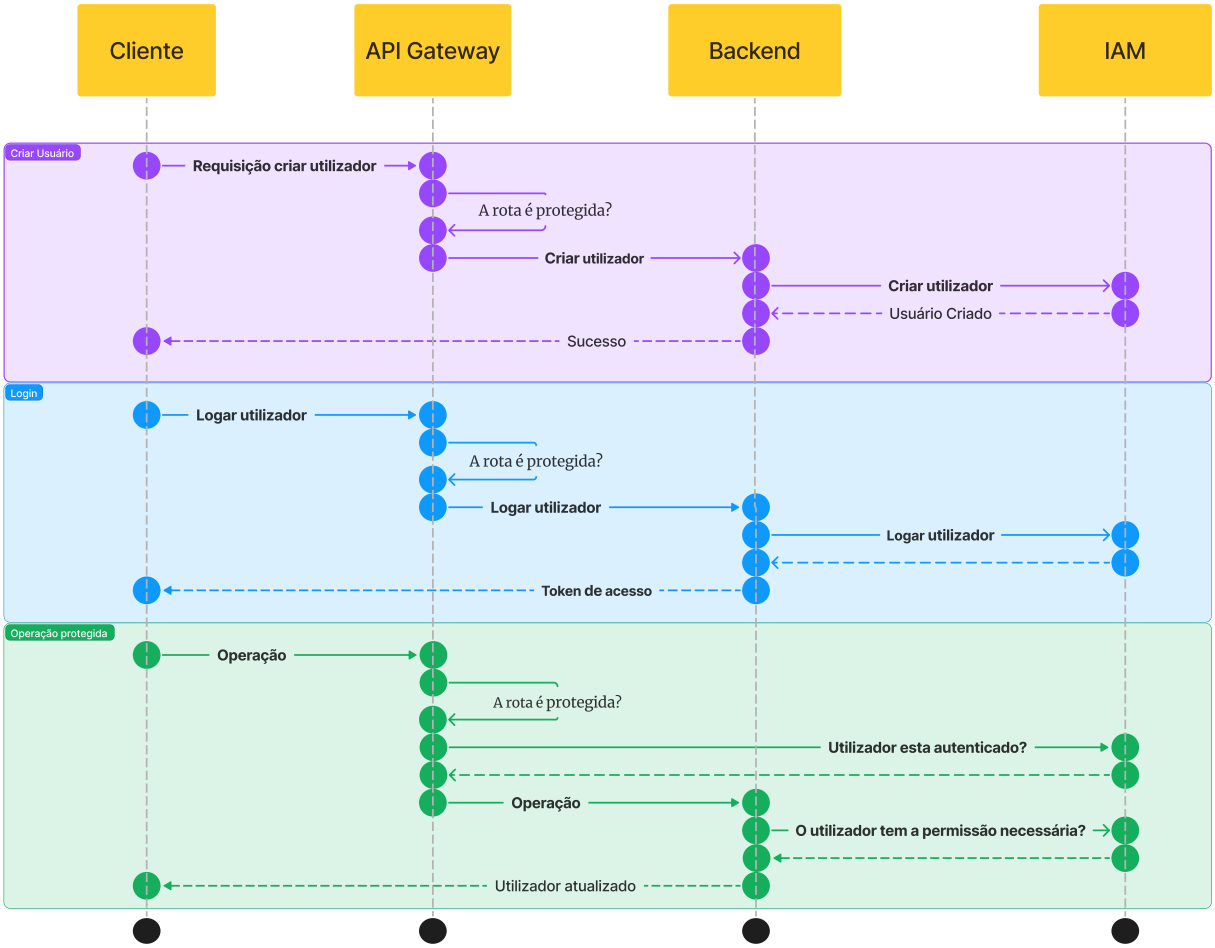


Figura 3.6: Diagrama de sequência de autenticação.

3.3.3 Gestão de GPUs

As GPUs são o componente de *hardware* mais importante no HaaS, pois são os dispositivos de computação com maior poder de aceleração do Hashcat, e costumam ser em número reduzido, o que implica um gerenciamento cuidada, para as rentabilizar o mais possível e, em simultâneo, servir as necessidades dos vários *containers* em execução em cada momento.

No arranque do *backend* os GPUs disponíveis são extraídos da listagem produzida pelo comando `clinfo -json`, que devolve um *output* em formato JSON, contendo a lista com todos os GPUs. Esta lista é filtrada, e as GPUs *online* são então extraídas e registadas no Redis (as GPUs *online* são as disponíveis no sistema, pois o `clinfo` também pode mostrar as GPUs *offline*, ou seja, que uma vez *online* foram removidas do sistema).

A partir daí, as GPUs inventariadas ficam em um estado indefinido, estratégia escolhida a fim de impedir a utilização de uma GPU conectada ao HaaS, antes sejam defina os modos de utilização explicitamente. É portanto necessária a atuação do administrador do sistema a fim de definir o modo de uso desses GPUs, que pode ser *partilhado* ou *exclusivo*.

Assim, uma GPU exclusiva pode ser usada por um só *container* Hashcat, em cada momento. Adicionalmente, um *container* não pode usar mais que uma GPU exclusiva.

Já uma GPU partilhada pode ser utilizada em simultâneo por vários *containers*. No entanto, é necessário que o utilizador que lança um *container* indique explicitamente quais as GPU partilhadas que quer usar, podendo não usar nenhuma.

Em suma, sob o ponto de vista de uma GPU individual: se for exclusiva, é usável por um e um só *container*, em cada momento; se for partilhada, é usável por mais que um *container* em simultâneo. Sob o ponto de vista de um *container*: pode usar, no máximo, uma GPU exclusiva, e pode usar qualquer número de GPUs partilhadas.

A limitação de uma GPU exclusiva por *container* Hashcat existe para otimizar o gerenciamento de recursos, facilitando o balanceamento de carga e ainda criando um freio para que o utilizador não escolha todas as GPUs disponíveis no sistema.

O conjunto de GPUs associadas pode ainda ser alterado a qualquer momento, desde

que as restrições descritas sejam respeitadas. Para que isto seja possível, foram implementadas programaticamente as descobertas realizadas nos experimentos iniciais. Assim, recorre-se ao *checkpoint* do Hashcat, descartando o *container* antigo. O arquivo de restauro é modificado para refletir o novo conjunto de GPUs, seguido do arranque de um novo *container* Hashcat, o qual deverá conter o arquivo de restauro modificado.

É também necessário, em termos de gerenciamento das GPUs, atualizar o registo da sua ocupação: desvincular do *container* destruído as GPUs que lhe estavam associadas, e vincular o novo conjunto de GPUs ao *container* criado .

Note-se que, em rigor, a eliminação do *container* anterior e a criação de um novo não seriam absolutamente necessárias, querendo-se manter o mesmo *host* para ambos; ou seja, nesse caso seria possível alterar apenas o conjunto de GPUs de um *container*, sem o recriar, usando apenas o mecanismo de *checkpoint*. No entanto, apesar das vantagens em termos de desempenho, esta abordagem possui menor flexibilidade em termos gerais. Por exemplo, a recriação do *container* permite a reutilização do código em diferentes situações, sem a necessidade de validações adicionais.

Papel do rOpenCL

Tanto o servidor (*backend*) do HaaS, como os *containers* Hashcat criados, têm acesso ao mesmo conjunto global de GPUs exposto pelo rOpenCL. Para tal, as imagens do *container* Hashcat e do *container* HaaS precisam ser construídas de forma equivalente, contendo ambas o rOpenCL corretamente configurado (ver Apêndice C).

O rOpenCL possui um arquivo de configuração que contém informações como interface de rede, endereço local e endereço do *host* dos GPUs remotos. Algumas destas informações em um *container* são alteradas sempre que se cria um *container*, outras devem ser definidas pelo administrador do sistema, e ainda existem algumas informações que são constantes.

Assim sendo, surge a necessidade de alterar o arquivo de configuração, sempre que um novo *container* for criado, de forma a garantir que a informação esteja correta. Para conseguir isso, criou-se um *script bash* (ver Figura 3.7), que executa à cabeça na inicialização de um *container* Hashcat. O papel do *script* é escrever no arquivo de configuração

```
1   #!/bin/bash
2   echo -e "1024\n0\n$(ifconfig eth0 | grep "inet" | awk '{print $2}' | cut
    -d/ -f1)\n1050\n$ROPENCL_HOSTS" > /etc/OpenCL/rOpenCL/config.txt
```

Figura 3.7: *Script* de Inicialização do rOpenCL.

do rOpenCL (nomeado como `config.txt`), várias definições, uma por linha:

1. **1024**: tamanho do *buffer* usado nas mensagens UDP;
2. **0**: valor booleano referente à utilização do UDP como protocolo de comunicação (0 = desativado, 1 = ativado).
3. **$(ifconfig eth0 | grep "inet"| awk '{print $2}'| cut -d/ -f1)**: obtém e escreve o endereço IP do *container* separando dos demais retornos;
4. **1050**: porta a usar para comunicar com as máquinas que têm GPUs;
5. **$ROPENCL_HOSTS**: lista dos endereços IP das máquinas que serão utilizadas.

A forma encontrada para inserir os *hosts* rOpenCL sem a necessidade de recompilar e construir uma nova imagem, foi através das adição através da variável de ambiente "$ROPENCL_HOSTS". Porém é necessário que o serviço do rOpenCL nos *hostd* esteja sempre a correr na mesma porta, conforme definido no arquivo de configuração.

A partir do rOpenCL, as GPUs podem ser utilizados de forma simplificada, abstraindo a sua localização. Uma outra vantagem é que as listagens, ordens e configuração dos GPUs aparecem da mesma forma, mesmo que em máquinas diferentes (pois o rOpenCL lista-as segundo a ordem que está no arquivo de configuração).

Com o suporte a vários modos de utilização dos GPUs, surge a necessidade de manter controle sobre a disponibilidade deles, de forma que o HaaS possa alocar ou negar o uso de GPUs exclusivas que já estiverem sendo utilizadas. Desta forma, é necessário saber quando a execução do Hashcat foi encerrada para que o GPU se torne disponível novamente. Para isso, é criado um *Observer* que é notificado pelo Docker sempre que um *container* é encerrado, podendo assim alterar o estado dos GPUs - ver Figura 3.8.

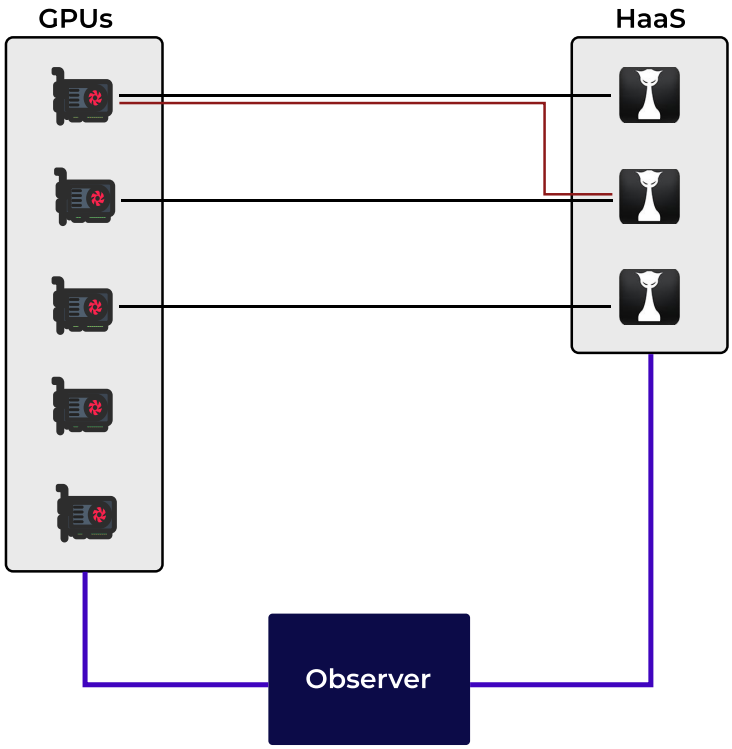


Figura 3.8: Gestão de GPUs.

O *Observer* inicia-se com a aplicação, e fica monitorizando as notificações do Docker, filtrando por *containers* Hashcat, e notificando os ouvintes (por meio de uma *chain*).

Para evitar *deadlocks* e permitir a consistência da aplicação, os comandos entre os serviços de GPUs e do Hashcat são realizados por meio de um *channel* (uma forma nativa do Go para comunicar e sincronizar *goroutines*).

3.3.4 Gestão do Hashcat

A criação de *containers* Hashcat envolve a criação de ficheiros num volume compartilhado entre o *container* e o *host* (e exportado via NFS), a criação da entidade na base de dados para armazenar todas as informações necessárias e o gerenciamento de GPUs.

Ao iniciar o processo de criação, cada GPU é alocado, sendo liberado somente a após o término da execução. Ainda no caso de GPUs em modo exclusivo, estas só podem ser alocadas caso estejam sem nenhuma utilização.

Antes da criação propriamente dita do *container*, é necessária a criação das suas configurações, onde as informações para inicialização do *container* devem ser preenchidas, a imagem selecionada, os volumes devem ter a ligação estabelecida e as labels definidas.

Após isso, o *container* pode ser criado. Este processo é realizado de forma assíncrona, pois pode demorar (o que acontece quando a imagem não está em *cache* e é necessário o *download*, ou havendo a necessidade de realizar *download* de alguma dependência atualizada, por exemplo) e causar um erro de *timeout*. Por isso ocorre em uma *thread* diferente, a requisição é respondida como sucesso e o estado do Hashcat definido como "em criação". Para a criação utiliza-se a biblioteca em Go do Docker, que permite interagir diretamente com o *socket* docker através da chamada de algumas funções.

Caso o mecanismo de *checkpoint* seja ativado, o *container* em questão é encerrado, mantendo-se armazenado todo o estado interno do momento em que o *checkpoint* foi realizado. Para o restauro é criado um novo *container* que irá partir do ponto de *checkpoint*.

Para que o utilizador do HaaS tenha o mesmo nível de controle da aplicação que o utilizador em linha de comando, foi necessária o redirecionamento do *input* e *output* do Hashcat. Esses redirecionamentos podem ser vistos na figura 3.9, onde o *input* e *output* representam, respetivamente, o comando de entrada para o Hashcat e o resultado da execução, e *Input Streaming* e *Output Streaming* representam as entradas de interação com o Hashcat, e os *logs* da aplicação, mostrando o progresso do programa.

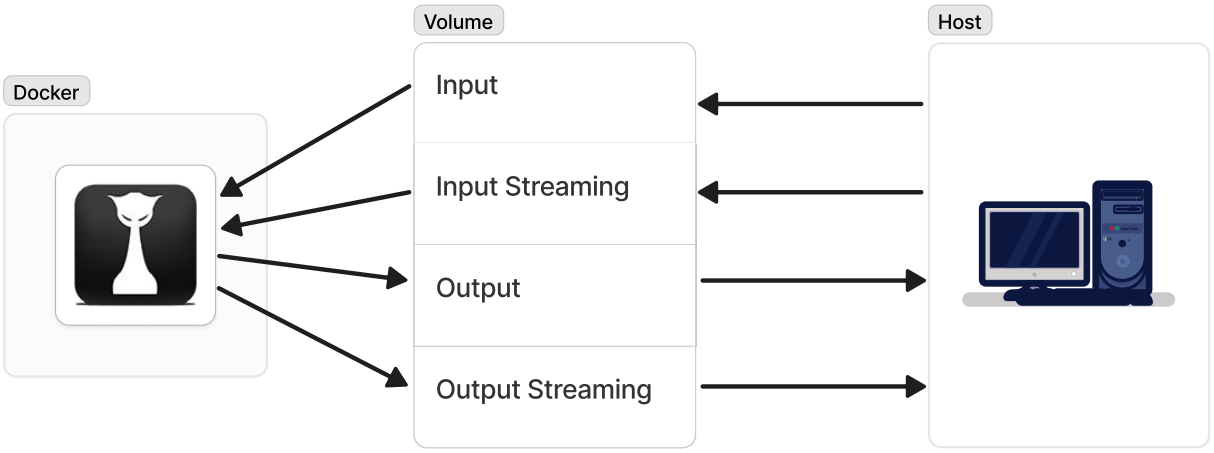


Figura 3.9: Arquivos de redirecionamento

Para o *input* é necessário que o arquivo de *input* seja constantemente monitorizado e suas entradas sejam redirecionadas à entrada do Hashcat, Para alcançar este resultado juntou-se a utilização de um *pipe* ligando o *output* do comando `tail` ao *input* do Hashcat.

De forma similar foi necessário redirecionar o *stdout* do Hashcat para um arquivo de *output*. Este arquivo pode posteriormente ser consultado a qualquer momento pelo utilizador do Hashcat e o seu conteúdo exportado como os *logs*.

Ao realizar os redirecionamentos descritos, o Hashcat pode ser utilizado em um *container* desanexado da sessão e consultado de acordo com a demanda do utilizador.

3.4 Mecanismo de Reserva

Para evitar que, caso não existam GPUs disponíveis, os demais utilizadores fiquem sem possibilidade de utilização da plataforma, criou-se um mecanismo de reserva. Através deste, caso um utilizador não encontre a GPU desejada disponível, poderá optar por reserva-lá; assim que estiver disponível, o *container* Hashcat para o qual a reserva foi feita, terá o seu conjunto de GPUs alterado automaticamente.

A implementação deste mecanismo de reserva apoiou-se nas funcionalidades de gerenciamento de GPUs já desenvolvidas, e no *Observer* criado para monitorizar e auxiliar na gerenciamento das GPUs. Foi apenas necessário estender o comportamento do sistema, adicionando um novo *Observer*, que entra em funcionamento assim que uma ou mais GPUs são libertadas. Este *Observer* analisa a fila das GPUs em reserva e realiza a atualização do conjunto (seguindo desta forma o comportamento descrito na secção 3.3.3).

Para a implementação desta funcionalidade, é necessário criar uma fila que gerencie as instâncias (*containers* Hashcat) aguardando pela GPU (ver Figura 3.10), além de adicionar um novo campo na requisição de criação, onde constam as GPUs a reservar.

O modelo de fila escolhido é o FIFO (First-In-First-Out). Nesta fila, a primeira instância a reservar a utilização da GPU é a primeira que terá acesso à GPU quando disponível.

É necessário criar uma fila para cada GPU. Tais filas precisam conter um identificador para a instância que fez a reserva, para que se possa atribuir-lhe a GPU quando disponível.

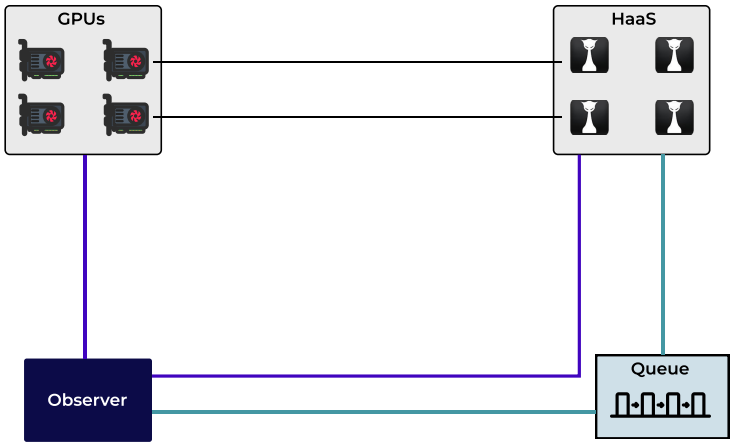


Figura 3.10: Gestão de GPUs com fila.

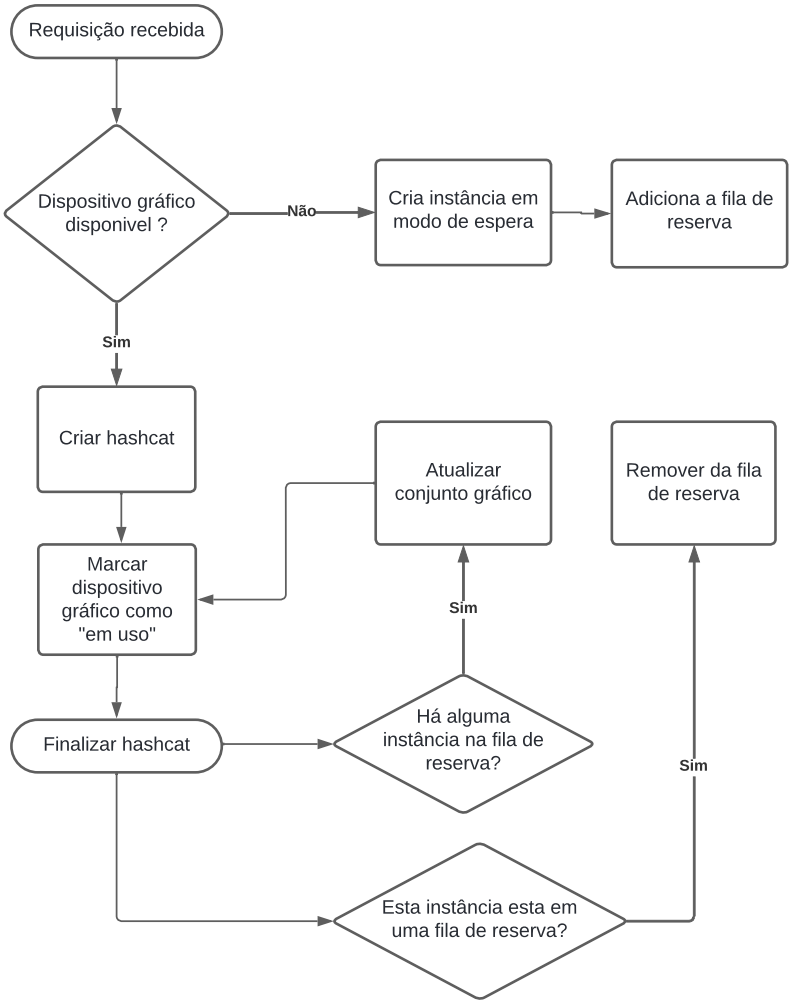


Figura 3.11: Fluxograma de reserva de GPUs.

O bimapeamento é necessário, pois, quando uma instância finaliza, é preciso removê-la da fila caso ainda não tenha chegado à primeira posição. A Figura 3.11 mostra o fluxo de reserva de GPUs, ilustrando quando uma instância é adicionado a / retirada de uma fila.

3.5 Operação em Cluster

Na descrição feita até agora, assumiu-se implicitamente que todos os serviços específicos do HaaS correm com Docker numa só máquina física. Apesar disso, tal não impede que GPUs remotas possam ser usadas, através do rOpenCL. No entanto, idealmente, um *container* Hashcat deveria estar co-localizado com as GPUs que oferecem mais desempenho, e como a disponibilidade e contenção sobre estas GPUs pode variar, é interessante ter a possibilidade de variar o local (máquina física) onde o *container* está alojado.

A integração do HaaS em um cenário de *cluster* acrescenta flexibilidade, desempenho e eficiência a toda a aplicação. Neste cenário, as instâncias criadas pelo HaaS passam a ser criadas como um serviço [69] e não mais como um *container* [70].

A diferença entre ambos é que um *container* é uma instância em execução de uma imagem Docker, enquanto que serviços são uma definição de como executar *containers*, incluindo a imagem Docker, a quantidade de réplicas, redes, volumes, e políticas.

A real vantagem deste cenário está na criação dos serviços, que pode ser feita dentro de qualquer nó do sistema, aumentando a flexibilidade e desempenho. Desta forma, torna-se possível que uma instância Hashcat seja sempre local às GPUs usadas de forma exclusiva.

3.5.1 Configuação do Docker Swarm

Para tirar proveito das vantagens acima citadas, é necessário um orquestrador de *containers*. Neste trabalho, a escolha feita é o Docker Swarm. Para isso, é necessário que toda a parte interna de comunicação com o Docker seja alterada para criação de serviços.

A utilização prematura do rOpenCL para acesso a GPUs locais e remotas afirmou-se como uma decisão arquitetural correta, pois abstrai a necessidade de realizar o mapeamento dos GPUs entre os nós a serem utilizados. Consequentemente, isso implica em uma

melhor organização, onde, devido à mesma ordem dos *hosts* nos arquivos de configuração do rOpenCL garantida programaticamente, são apresentados e utilizados na mesma ordem, não necessitando de nenhum tradutor entre nós diferentes.

O diretório de compartilhamento de arquivos também necessita de uma atualização para garantir que todos os nós do conjunto estejam incluídos no compartilhamento do diretório, evitando a movimentação de arquivos entre diferentes nós.

Esta escolha pela utilização de múltiplos nós cria a necessidade de modificar o mecanismos de eventos, responsável por dizer quando uma instância terminou. Esta modificação é necessária pois o Docker notifica as alterações realizadas apenas na máquina local; como é necessário saber os eventos em todos os nós, esta abordagem precisa ser modificada.

A solução encontrada foi a criação de um *loop* de eventos, que fica constantemente analizando o estado das instâncias em execução, encerrando o serviço e disparando um evento quando as instâncias finalizam.

3.5.2 Migração

A implementação do lançamento de instâncias via Docker Swarm, como descrito acima, possibilita a criação de um mecanismo de migração, permitindo que as instâncias do Hashcat passem a ser lançadas em um nó e, caso outro nó ofereça melhor desempenho, possam ser migradas para lá sem grande penalização em termos de tempo, melhorando ainda o desempenho geral da aplicação.

O mecanismo de migração foi limitado ao cenário de uma alteração do conjunto de GPUs, pois de modo simplificado é a melhor métrica de aumento de performance. Sendo assim, uma instância só migrará entre os nós disponíveis caso haja uma troca de GPU. Este processo é feito de forma automatizada e transparente ao utilizador da aplicação.

A migração tira proveito do escalonador do Docker Swarm, que já possui atributos que podem sugerir onde um determinado serviço pode ser colocado. Por exemplo, utilizam-se as *labels* de cada nó para definir quais as GPUs ali definidas como locais.

Combinando as *labels* presentes, inseridas ao se criar cada nó, e as configurações de

criação de cada serviço, é feita uma busca entre os nós definidos para lançar o serviço no nó com o melhor desempenho possível. Este processo repete-se se se alterar o conjunto de GPUs, fazendo com que a instância tenha sempre o melhor rendimento possível.

3.6 Interface com o Utilizador

A interface gráfica do HaaS representa a porta de acesso dos utilizadores à plataforma, sendo um interface de tipo web cujo desenvolvimento assentou no *framework* Next [71].

A interface gráfica permite criar utilizadores, *login*, gerenciar instâncias Hashcat e GPUs. Algumas destas funcionalidades são aqui apresentadas (para uma vista geral da interface e explicação com mais detalhe dos componentes, ver o Apêndice B).

A listagem de GPUs (ver Figura 3.12), apresenta as GPUs do sistema e informações relevantes, como o número de *containers* Hashcat utilizando-as, e a quantidade de reservas que a GPU possui. Permite ainda modificar os modos de uso.

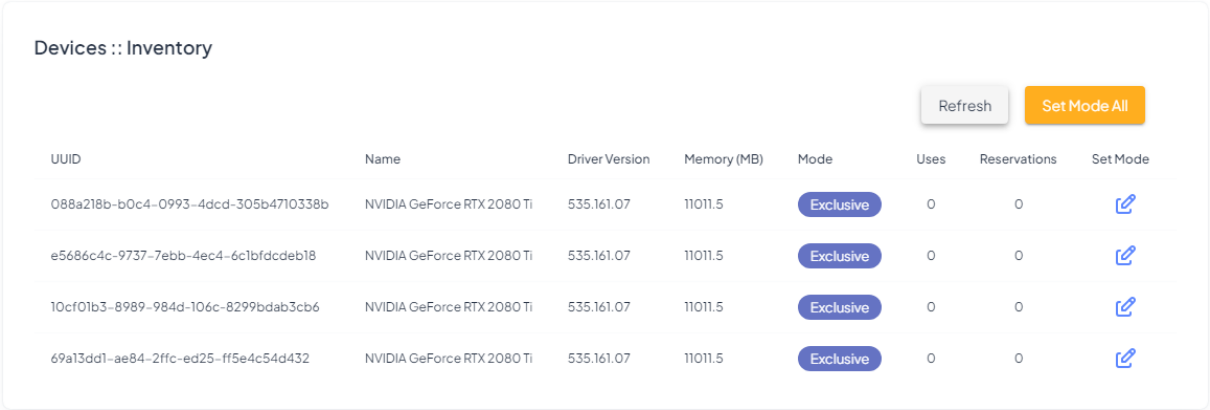


Figura 3.12: Listagem de GPUs.

Com o conhecimento da estado das GPUs do sistema, o utilizador pode optar pela instanciação de um *container* Hashcat (Figura 3.13), tendo que definir um nome (que funcionará como uma alias, para melhor identificação do utilizador) e as opções de utilização do Hashcat (modo de ataque, tipo de hashes a serem utilizados, as GPUs a se utilizar ou reservar e os *inputs*(que podem ser inseridos através de um arquivo ou texto simples).

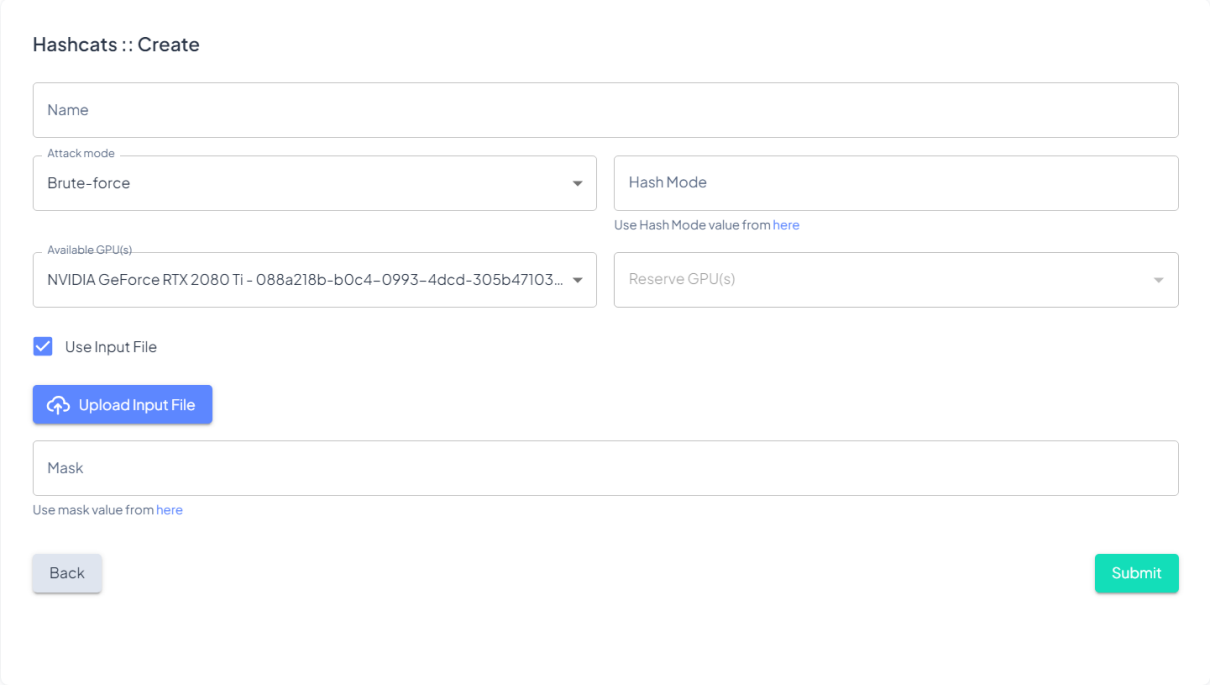


Figura 3.13: Formulário de Criação do Hashcat.

Os dados inseridos serão validados pela interface antes da submissão. Assim sendo, a inconsistência de informações ao submeter o formulário retornará erros de validação que serão apresentados ao utilizador e impedirão a criação do *container* Hashcat.

Após a criação do *container*, o utilizador poderá acompanhar o funcionamento do Hashcat através da interface web, como mostrado na Figura 3.14: é possível acompanhar os *logs*, visualizar o estado e interagir com o Hashcat em funcionamento.

O sistema garante ainda a consistência da aplicação, desativando algumas operações de acordo com o estado. Como visto na imagem, ao se ter finalizado a execução, não é possível ter acesso a nenhuma operação do Hashcat. O interface apresenta comportamento similares noutros estados (por exemplo, não é possível pausar a aplicação em *checkpoint*).

Para este controle da interface gráfica é necessário ainda o gerenciamento de estado interno no qual a interface deve manter as informações do utilizador e da aplicação. Para tal é utilizado o Redux [72], que armazena as informações e permite passá-las entre componentes, alterando o estado da aplicação a nível global com poucas operações.

Também é necessário notificar o utilizador a respeito das operações realizadas e se

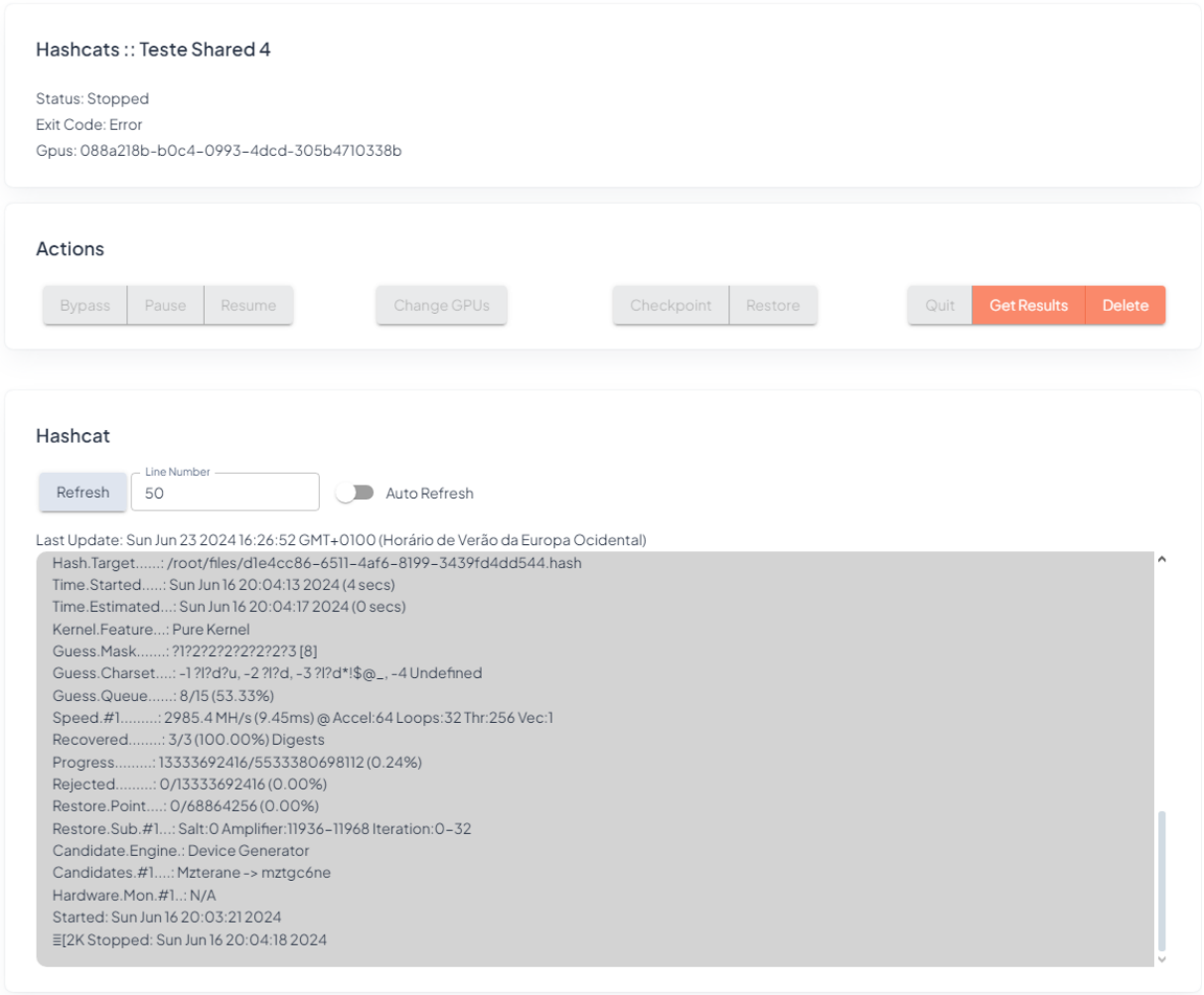


Figura 3.14: Visualização do *Container* Hashcat.

foram realizadas com sucesso ou erro. Para isso foi implementado um notificador que exibe um barra contendo os alertas. Esta barra foi implementada como um componente global, podendo então ser exibida através da atualização de estado da aplicação.

Além da notificação através do alerta, a notificação de erros em formulários é feita através de *labels* contidas no formulário, notificando a respeito de erros no preenchimento antes que o formulário seja submetido.

Finalmente, antes de renderizar cada componente protegido, a interface gráfica verifica o *token* JWT e as autorizações do utilizador; caso o utilizador não tenhas as autorizações devidas, o componente não será renderizado e este não poderá ser acessado (por exemplo,

caso o utilizador não possua credenciais administrativas, não visualizará opções de atualizar o modo de uso presentes na Figura 3.12). Além disso caso o utilizador não esteja autenticado, ou caso o *token* não seja válido, o utilizador é redirecionado para a página inicial, onde pode optar por se autenticar novamente.

3.7 Comunicação

A ligação entre a interface gráfica e o *backend* deu-se através de uma API REST, gerenciada através do Kong que inspeciona quaisquer requisições, direcionando para o controlador correto e fazendo validações de autenticação antes de redirecionar.

Na autenticação do utilizador, o cliente web armazena o *token* de autenticação internamente, enviando-o em cada requisição feita em seguida.

As rotas disponíveis abrangem todas as funcionalidades que foram vistas até agora, sendo usadas durante toda a interface gráfica. A implementação de cada rota pode ser vista com mais detalhe em [68].

3.8 Implementação Atual

Atualmente, a plataforma HaaS permite realizar as seguintes operações, seja através do interface com o utilizador, seja de forma programática (através de uma API interna):

- **Utilizador**
  - **Cadastro**: Realiza o cadastro na aplicação, através da submissão das informações de acesso e conta.
  - **Login**: Realiza o login do utilizador na plataforma, validando as credenciais e retornando um *token* que autoriza o utilizador a acessar as demais rotas.
- **GPUs**
  - **Listagem (Protegida)**: Lista os GPUs disponíveis, dando detalhes como o número de utilizadores, modo de execução, entre outros.

  - **Alteração do modo de execução (Protegida)**: Muda o modo de execução dos GPUs; esta funcionalidade somente pode ser utilizada por um utilizador com permissões administrativas.

- **Hashcat**:
  - **Criação (Protegida)**: Criação da instância com o Hashcat.
  - **Listagem (Protegida)**: Listagem das instâncias criadas; as instâncias retornadas se limitam às instâncias criadas pelo utilizador; além disso as informações são retornadas em uma forma resumida.
  - **Detalhes (Protegida)**: Exibe os detalhes de um instância.
  - **Status (Protegida)**: Retorna os logs de execução do Hashcat; permite ainda escolher a quantidade de linhas que será retornada (por padrão, 150).
  - **Operações (Protegida)**: Operações de modificação da instância do Hashcat:
    * Resume: Retoma a operação do Hashcat.
    * Pause: Suspende (temporariamente) a operação do Hashcat.
    * Bypass: Avança para o próximo ataque sem terminar o atual.
    * Checkpoint: CCria um arquivo de restauro no próximo ponto de paragem, e encerra o Hashcat.
    * Quit: Encerra o Hashcat de imediato.
    * Restore: Reinicia a operação do Hashcat a partir de um arquivo de restauro.
  - **Remover (Protegida)**: Remove a instância e todos os arquivos e dados associados.
  - **Download de resultados (Protegida)**: Faz download do arquivo de resultados da instância; não é necessário que a execução termine para estar disponível.
  - **Migração (Protegida)**: Permite a atualização dos GPUs da instância, realizando a migração entre nós.

De referir, também, algumas restrições importantes da implementação atual:

- Cada *container* Hashcat pode ter associada apenas uma GPU exclusiva; esta simplificação foi considerada necessária, pois ao se incluir mais de uma GPU, o escalonador e o processo de migração passariam a possuir uma complexidade maior, fora do escopo deste projeto.
- Um utilizador pode selecionar para um *container* apenas GPUs exclusivas, pois as GPUs partilhadas são automaticamente associadas a todas as instâncias que optam pela utilização de GPUs partilhadas (portanto, não é possível selecionar um subconjunto das GPUs partilhadas: ou se usam todas, ou nenhuma);
- Apenas GPUs em modo exclusivo possuem um mecanismo de reserva; a extensão desse mecanismo a GPUs partilhadas faria sentido a fim de evitar *oversubscribing*.

Desta forma, a implementação atual do HaaS suporta uma utilização funcionalmente abrangente, permitindo que os utilizadores realizem um leque variado de operações no âmbito da quebra de senhas com base na ferramenta Hashcat, ao mesmo tempo que partilham as GPUs disponíveis ou, caso desejem, as utilizam em exclusividade.

Capítulo 4

Testes e Avaliação

Este capítulo apresenta os testes realizados para avaliar o comportamento do HaaS em diversos cenários de uso. Sendo o HaaS uma plataforma voltada para a execução de cargas de trabalho intensivas em GPU, é essencial entender como diferentes configurações e modos de utilização impactam o desempenho e a estabilidade da plataforma.

Começa-se com uma descrição das premissas que guiaram os testes, destacando a importância de garantir que os mesmos são realizados em ambiente controlado. Em seguida, analisam-se os resultados obtidos em diferentes modos de execução - partilhado e exclusivo -, observando-se como o tempo de resposta e a eficiência das GPUs variam com base na configuração escolhida.

Abordam-se ainda limitações identificadas, especialmente aquelas relacionadas ao uso de GPUs remotas e os desafios impostos pelas restrições de rede. A partir das observações feitas, discute-se como a flexibilidade do HaaS permite ajustar dinamicamente a utilização das GPUs para melhor atender às necessidades específicas das cargas de trabalho.

Por fim, a análise culmina em uma discussão sobre a adaptabilidade do sistema e as recomendações para otimizar o desempenho, considerando as variáveis críticas que influenciam a eficiência das operações. Esta avaliação fornece uma base sólida para futuras melhorias e adaptações do HaaS, assegurando que ele possa oferecer um desempenho robusto e eficiente em uma ampla gama de aplicações.

4.1 Ambiente Computacional

O ambiente de testes consistiu em duas máquinas virtuais, alojadas num *cluster* de virtualização Proxmox 7 do CeDRI/IPB, com configuração idênticas de hardware e software. Cada máquina virtual foi alojada num nó diferente e idêntico do *cluster*. Em termos de hardware, cada máquina virtual possui a seguinte configuração comum: 32 núcleos de uma CPU AMD EPYC 7452 2.4/3.4Ghz, 64 GB de RAM ECC DDR4 2666 MHz, 512 GB de disco virtual assente em SSD NVMe PCIe 3 U.2, placa de rede virtual (Linux bridge) assente num adaptador físico Mellanox ConnectX-5 EN 100 Gbps Ethernet, duas GPUs NVIDIA RTX 2080 Ti 11GB expostas via *passthrough*.

Além disso, cada máquina possui o mesmo ambiente operacional (sistema operativo e versões das ferramentas e plataformas de base usadas), destacando-se: sistema operativo Linux Ubuntu 22.04 de 64 bits, OpenCL 3.0, rOpenCL 1.1, Hashcat 6.2.5 e Docker 25.0.3.

Os softwares dos quais o HaaS depende são gerenciados através do Docker usando a facilidade Docker Compose. Esses softwares incluem PostgreSQL, Go, Redis, Kong, Keycloak, entre outros. Um software que possui grande impacto na realização dos testes é o Docker Swarm, que conecta ambas as máquinas virtuais em um sistema distribuído.

4.2 Metodologia e Benchmarks

A avaliação baseou-se na execução, em diferentes condições, de *benchmarks* que consistem em três cargas de trabalho idênticas, que têm como objetivo quebrar as mesmas cifras (sendo por isso configuradas da mesma forma em termos de hashes, algoritmos e modos de execução do sistema HaaS). Sendo certo que num cenário real de utilização são expectáveis cargas de trabalho diferentes, a opção por cargas idênticas para *benchmarking* fornece um ambiente de avaliação uniforme, que facilita a extração de conclusões.

De referir que o número de cargas de trabalho é três por ser esse o número máximo simultâneo, em modo partilhado, permitido pela memória disponível nas GPUs usadas, considerando os algoritmo e a parametrização específica usada no Hashcat (ver a seguir).

Cabe ainda esclarecer que o conceito de *carga de trabalho* corresponde, no contexto desta tese, a uma instância da aplicação Hashcat em execução dentro de um *container* dedicado a esse fim, pelo que o conceito de carga de trabalho e *container* Hashcat confudem-se, usando-se de forma indistinta.

Os *benchmarks* foram feitos no modo de força bruta do Hashcat, utilizando 20800 como parâmetro de hash [1], onde as senhas são primeiro submetidas ao algoritmo MD5 [73] e, em seguida, ao SHA256 [74]. As senhas e respetivos hashes escolhidos foram:

- Test1234 : 52bb65a50112c1d8c2836c3cda4a1a6338dfce0c17c1f5c9fa57181d504a15e5
- Password : e765e47fd32132c38dd4826f75556fc0e9038d775bf705f76c147b352362695b
- P455w0rd : 65e9ae1761e1473afe9cdcac1ea9ceeca2f1346093621c6fac20f032c159813d

A escolha adequada das senhas e algoritmos é de grande importância. Se forem muito simples não produzem tempos de execução relevantes para o *benchmark* e não sofrem impacto significativo da intermediação do HaaS. Por outro lado, cifras e senhas muito complexas tornam o tempo dos testes e o consumo de recursos muito difíceis de serem analisados, podendo ser afetados por fatores externos e mais difíceis de monitorar.

Para cada *benchmark* foram registadas várias métricas, para uma avaliação mais abrangente e precisa do desempenho da aplicação HaaS em diferentes cenários de uso: latência no arranque e terminação dos *containers* Hashcat, tempo de execução das instâncias Hashcat, nível de utilização e consumo energético das GPUs utilizadas.

Os diferentes cenários testados envolveram diferentes combinações de GPUs em que o número de GPUs combinadas vai de 1 a 4, localização (local e/ou remota) e modos de utilização (exclusivo vs partilhado) das GPUs empregues. Cada configuração resultante de uma destas combinações foi testada quatro vezes. Os resultados apresentados são relativos às últimas três repetições do teste.

A avaliação de cada cenário foi despoletada de forma programática, com base em código assente na API da aplicação desenvolvida – ver exemplo de *script* de *benchmarking* no Apêndice E – e os resultados extraídos manualmente dos *logs*, sendo ainda comparados

com as informações na base de dados. Esta abordagem facilitou a repetição dos testes sempre que necessário. Permite ainda assegurar a sua reprodutibilidade, fundamental para que terceiros verifiquem, de forma independente, a validade dos resultados obtidos.

4.3 Avaliação Inicial

Antes da avaliação exaustiva do HaaS na variedade de cenários que emergem ao combinar os principais parâmetros experimentais, efetuou-se um conjunto inicial de testes para aferir algumas métricas básicas e permitir simplificar a análise dos testes posteriores.

4.3.1 Tempo de Vida de um *Container*

A primeira ronda de testes consistiu em medir e comparar o desempenho global propiciado pelos dois modos de utilização de GPUs (exclusivo e partilhado) do HaaS em um cenário básico: uma única carga de trabalho (*container* Hashcat) é executada, numa só máquina virtual, usando uma só GPU (local, em modo exclusivo ou partilhado). Este desempenho é aferido pelo tempo total de vida, ou simplesmente *tempo de vida* dos *containers*: tempo que decorre desde o pedido de criação de um *container*, até que este é dado por destruído.

O tempo de vida de um *container* Hashcat inclui assim: a) *latência de arranque* – tempo desde o pedido de criação do *container* até ao arranque da aplicação Hashcat no mesmo; b) *tempo de execução* – tempo desde o arranque da aplicação Hashcat no *container*, até ao término dessa aplicação; c) *latência de destruição* – tempo desde o término da aplicação Hashcat no *container*, até à destruição (remoção) desse *container*.

Recapitulando ainda que as GPUs em modo exclusivo são aquelas utilizadas por um e apenas um *container* Hashcat, as quais se limitam a apenas uma por *container*, enquanto que as GPUs em modo partilhado não restringem a quantidade de *containers* a utiliza-lás.

O gráfico da Figura 4.1 apresenta o resultado deste teste. Os valores apresentados são médias, para cada modo, das últimas 3 de 4 execuções.

Como se pode verificar, os tempos de vida são similares. Mas em rigor, há uma muito ligeira vantagem para o modo partilhado, pois no modo exclusivo há algum código

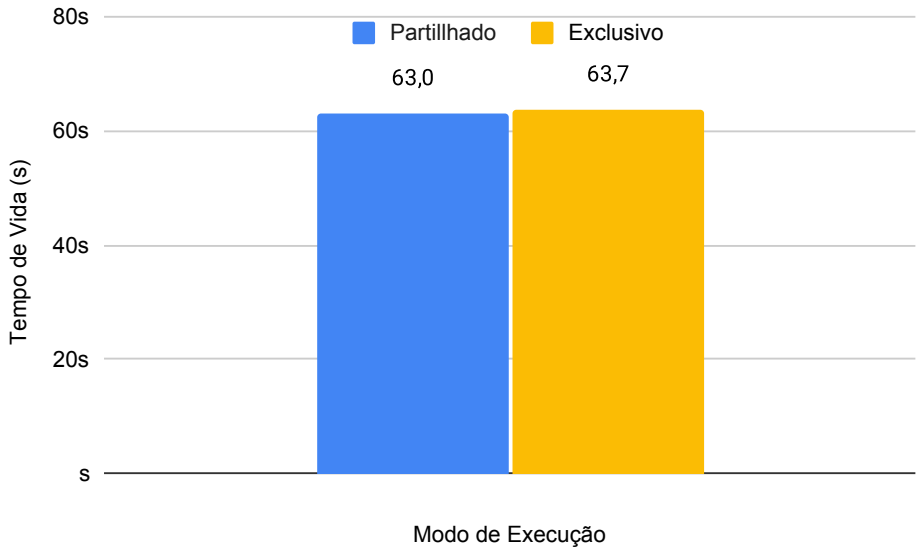


Figura 4.1: Tempos de vida de um *container* Hashcat com 1 GPU (local).

adicional que tem de ser executado para garantir a exclusividade do uso da GPU. Mas, fora isso, como neste caso não há concorrência no uso da GPU por mais que um *container*, o modo de execução não afeta praticamente o desempenho.

Isto permite antecipar que as eventuais diferenças mais marcadas de desempenho apresentados pelos dois modos, em cenários mais complexos, terão outras causas, que não internas aos modos (como por exemplo um modo ter sido implementado de forma mais eficiente que o outro, o que introduziria um enviesamento artificial dos resultados).

4.3.2 Latência de Arranque com vários *Containers*

A segunda ronda de testes iniciais destinou-se a avaliar a *latência de arranque* quando é solicitada a criação de vários *containers* Hashcat, também na presença de uma só GPU. Recorde-se que para cada *container*, esta latência refere-se ao tempo que decorre entre o pedido de criação desse *container* e o arranque da aplicação Hashcat nesse *container*.

Nestes testes, a única GPU disponível é usável nas seguintes configurações: quando usada em modo exclusivo, será local; se for usada em modo partilhado, poderá ser local ou remota; se for local apenas estará envolvida uma máquina virtual; se for remota serão usadas as duas máquinas virtuais do ambiente de teste, neste caso com os serviços HaaS

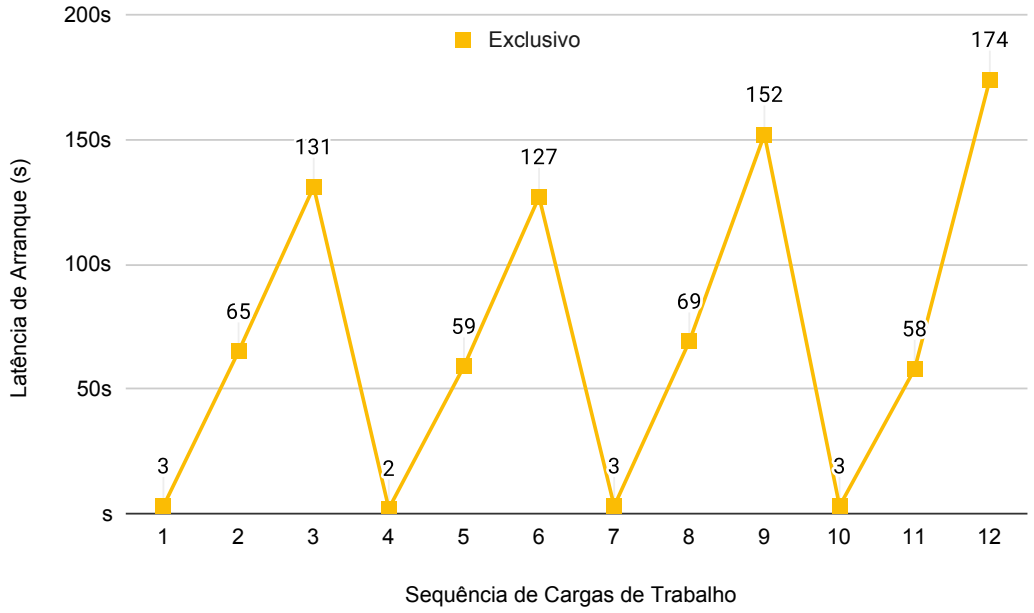
a correr apenas numa máquina, e os serviços rOpenCL a correr nas duas.

Estes testes foram realizados em 4 rondas seguidas, repetindo-se 4 vezes para cada modo de uso da GPU (local partilhada, remota partilhada, local exclusiva). Em cada ronda, foram lançados três *containers* Hashcat, registando-se a latência de arranque de cada um. Os resultados estão apresentados na Figura 4.2, em separado para os modos exclusivo (Figura 4.2a) e partilhado (Figura 4.2b), dadas as ordens de grandeza das latências. Os pontos 1 a 3 do eixo horizontal representam as 3 cargas (*containers* Hashcat) da primeira ronda, os pontos 4 a 6 representam as da segunda ronda, e assim sucessivamente.

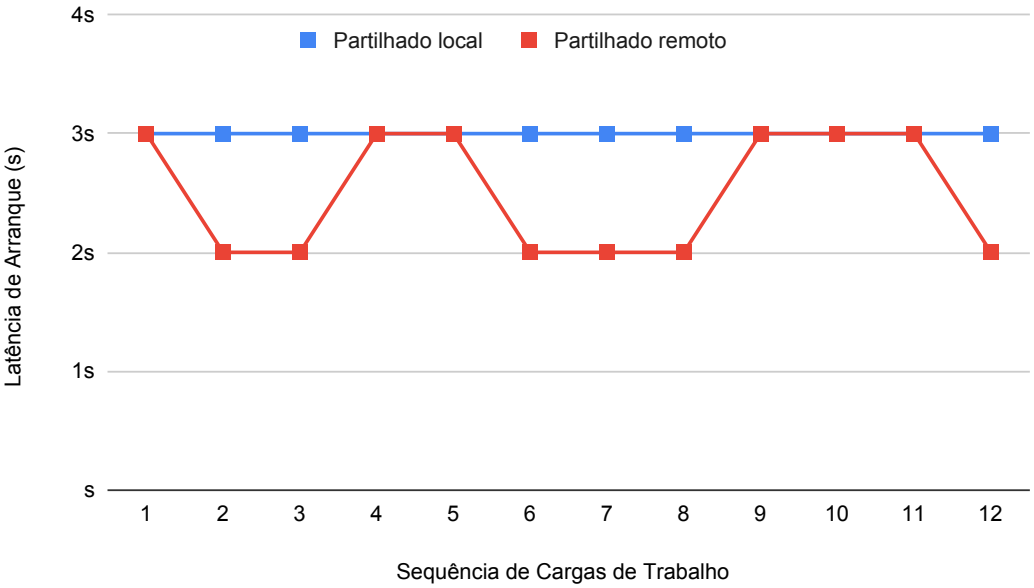
Observa-se uma disparidade significativa entre os valores obtidos nos modos exclusivo e partilhado. Essa diferença decorre do fato de que, enquanto no modo partilhado os *containers* Hashcat são criados imediatamente após a receção da requisição, no modo exclusivo cada requisição tem de aguardar que a GPU fique disponível, o que só acontece após a finalização da carga de trabalho anterior que a estava a utilizar. Esse *tempo de espera* resulta na variação temporal encontrada, que era aliás esperada, devido à fila de espera usada para gerenciar as GPUs em modo exclusivo.

Para perceber o tempo que é efetivamente gasto, no modo exclusivo, no lançamento e arranque de um *container* Hashcat, é necessário subtrair da *latência de arranque* o *tempo de espera* (na fila da GPU), resultando numa *latência de arranque ajustada*. A Figura 4.3 mostra essa latência ajustada para o modo exclusivo, junto com as latências originais do modo partilhado. Na perspetiva oferecida pela Figura 4.3, as *latências de arranque* dos 3 modos são afinal comparáveis, embora com uma tendência para uma maior latência no modo exclusivo, devido à necessidade de gerenciar a fila associada à GPU.

Com base nesta análise, conclui-se que o tempo gasto na criação de *containers* Hashcat é relativamente independente da sua configuração. Em relação ao tempo gasto na destruição (*latência de destruição*), embora não se apresentem valores, os mesmos são também relativamente consistentes para os vários modos de execução e localidades das GPUs. Desta forma, essas duas latências acabam por não ser relevantes para a avaliação de desempenho (e consequente *ranking*) das várias configurações, sendo por isso ignorados nos próximos benchmarks, os quais se concentram apenas no *tempo de execução*.



(a) Latências de arranque no modo exclusivo.



(b) Latências de arranque no modo partilhado.

Figura 4.2: Latências de arranque com vários *containers* Hashcat (1 GPU).

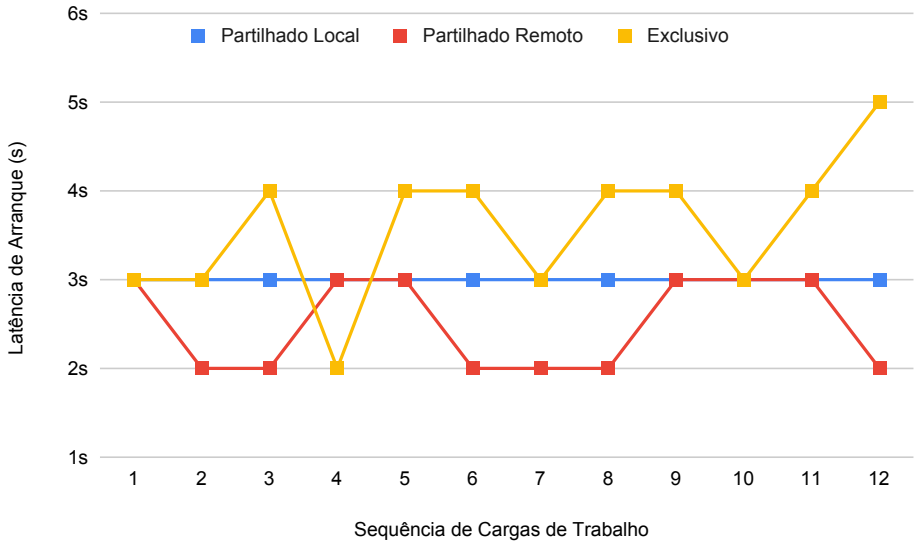


Figura 4.3: Latências de arranque com vários *containers* Hashcat (excluindo tempo de espera na fila da GPU para o modo exclusivo).

4.4 Avaliação do Tempo de Execução

Concluída a avaliação preliminar, partiu-se para uma avaliação geral, em todas as configurações permitidas pelo ambiente de testes e pelas restrições da arquitetura do HaaS.

Inicialmente, realizaram-se testes com os serviços HaaS em um só nó (máquina virtual), e os serviços rOpenCL presentes nos dois nós do ambiente computacional. Isso foi suficiente para uma avaliação com GPUs num só modo de execução (partilhado ou exclusivo), e com diferentes localidades (só locais, só remotas, ou ambas – secção 4.4.1). Depois, consideraram-se cenários híbridos no que respeita ao modo de execução, em que o modo partilhado e o exclusivo são usados em simultâneo (secção 4.4.2). Por fim, considerou-se a execução dos serviços HaaS em dois nós, possibilitando a avaliação dos mecanismos de migração, e seu impacto no desempenho do HaaS (secção 4.4.3).

De recordar a restrição de uso de não mais que uma GPU exclusiva por *container* Hashcat, como referido antes (rever secção 3.8). Essa restrição existe por motivos de otimização e gerenciamento de recursos. Como resultado, não faz sentido testar cenários com mais de três GPUs exclusivas, pois qualquer GPU adicional nesse modo não seria aproveitada, dado o número máximo admissível de *containers* Hahshcat em execução simultânea

ser também de 3 (rever secção 4.2). Portanto, esses cenários não foram contemplados.

De recordar ainda que se um *container* Hashcat usar uma GPU exclusiva, a mesma terá de ser local. Desta forma, em cada nó do ambiente de testes poderão co-existir, no máximo, 2 *containers* que usam GPUs exclusivas, dado haver 2 GPUs por cada nó. Complementarmente, como há 2 nós, com 2 GPUs cada, poderiam co-existir, no máximo, 4 *containers*, cada um com a sua GPU exclusiva. No entanto, por razões já referenciadas, o número máximo de *containers* simultâneos admitido nos testes é de 3.

Além disso, quando dois nós do HaaS são utilizados, surge alguma imprevisibilidade na alocação das GPUs: a única garantia é que as GPUs exclusivas serão locais, enquanto as GPUs partilhadas podem estar em qualquer nó. Isso pode afetar a interpretação dos resultados, especialmente em termos de latência e comunicação entre nós.

4.4.1 1 serviço HaaS, 1 Modo de Execução

GPUs Locais Partilhadas ou Exclusivas

Os primeiros cenários avaliados são os mais simples: apenas se usam GPUs locais e, quando mais de uma GPU é usada, todas o são no mesmo modo de execução. Assim, a Figura 4.4 apresenta os *tempos de execução*, em conjunto ou individualmente, das 3 cargas de trabalho (*containers* Hashcat), quando partilham 1 ou 2 GPUs locais. E, na Figura 4.5, podem-se ver os tempos correspondentes ao uso de 1 ou 2 GPUs em modo exclusivo.

Em cada figura, há dois tipos de tempo: um global, e um individual (ambos médias das últimas 3 de 4 repetições do teste). O tempo global é o tempo que decorre entre o instante em que se inicia a execução do Hashcat no primeiro *container* a iniciá-la, e o instante em que termina no último a terminá-la (podendo esses *containers* ser diferentes); é portanto o tempo de execução do conjunto dos 3 *containers*. O tempo individual corresponde ao valor médio do tempo de execução na perspetiva de cada *container*: tempo que decorre desde que o *container* inicia a execução do Hashcat, até que a termina; este tempo é sempre inferior ao tempo global (embora próximo deste no modo partilhado).

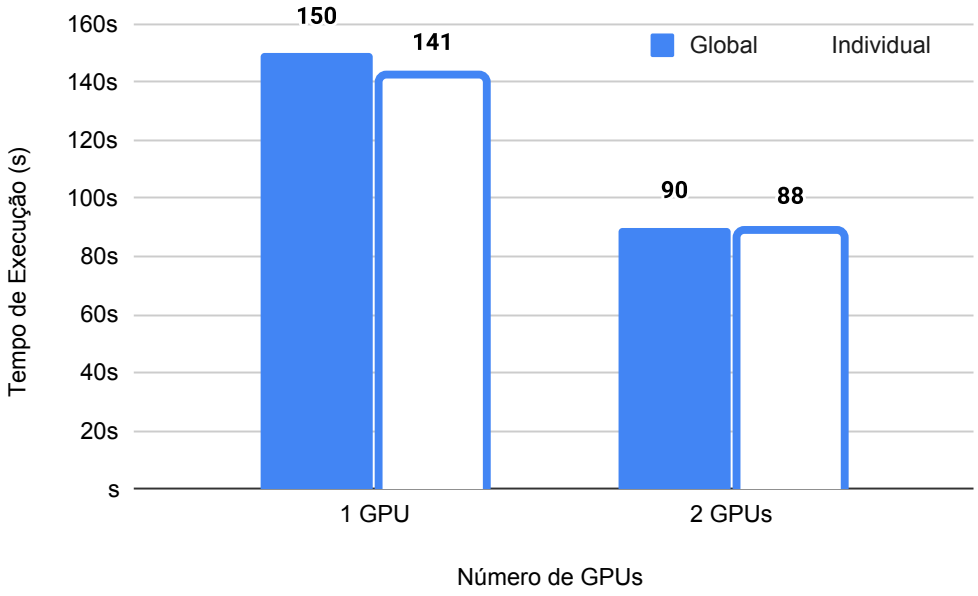


Figura 4.4: Tempos de execução com GPUs locais partilhadas.

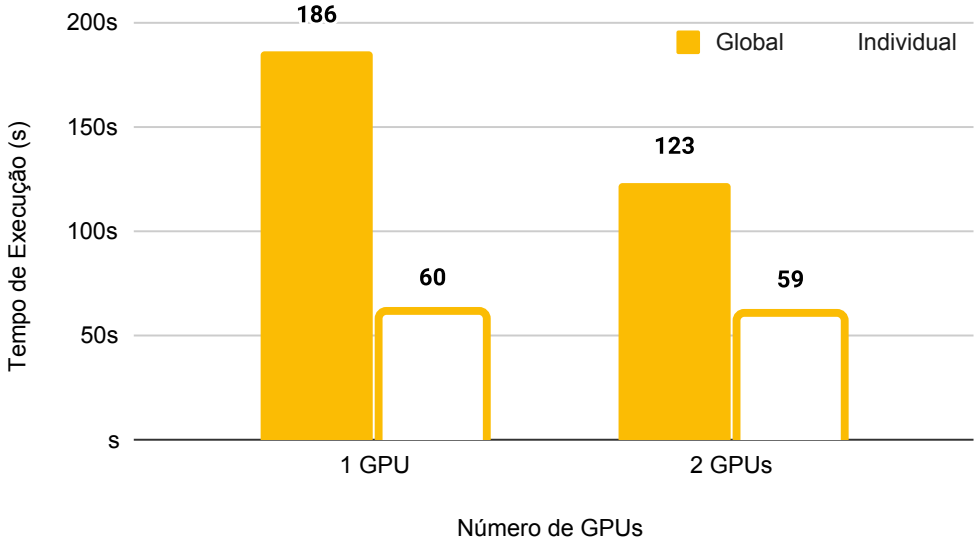


Figura 4.5: Tempos de execução com GPUs locais exclusivas.

Analisando e comparando os tempos globais nas duas figuras, conclui-se que:

- em ambos os modos de execução, adicionar mais uma GPU local provoca uma diminuição do tempo de execução global (o que era à partida esperado); e, embora em termos absolutos os tempos sejam diferentes consoante o modo, as acelerações são próximas (com ligeira vantagem para as GPUs partilhadas): *speedup* de 150/90 = 1.67 com duas GPUs partilhadas, e 186 / 123 = 1.51 com duas GPUs exclusivas.
- o uso de GPUs em modo partilhado permite uma execução mais rápida do conjunto das 3 *workloads*, face ao uso das GPUs em modo exclusivo: *speedup* de 186/150 = 1.24 com uma GPU envolvida, e de 123/90 = 1.37 com duas GPUs envolvidas.

O facto do uso das GPUs em modo partilhado oferecerem um menor tempo de execução do conjunto dos 3 *containers*, que as GPUs em modo exclusivo, foi inicialmente surpreendente e considerado contra-intuitivo, pois uma das premissas para a criação do modo exclusivo era a de que isso poderia favorecer uma execução mais rápida. Na realidade, atentando nos valores dos tempos Individuais, assim é: sob o ponto de vista individual, cada *container* executa mais rápido no modo exclusivo do que no modo partilhado. O que explica então que, para o conjunto dos 3 *containers*, seja mais rápido o uso do modo partilhado ?

Para responder a esta questão é necessário analisar também a carga de utilização de cada GPU nestes diferentes cenários, presente na Figura 4.6. Assim, é evidente que em modo exclusivo uma GPU é usada de forma menos eficiente, tendo folga para acomodar mais trabalho. Assim, partilhando a GPU por vários *containers*, o conjunto destes terminará mais rápido, em comparação com a sua execução sequencial em modo exclusivo.

Relativamente aos tempos Individuais, pode-se também observar que a adição de mais GPUs em modo exclusivo não altera esses tempos (60s vs 59s): uma vez que as GPUs são iguais entre si, é irrelevante que uma *workload* execute numa GPU ou noutra, sendo que a *workload* tem a GPU inteiramente disponível para si, e portanto os tempos de execução são similares. Já em modo partilhado, a disponibilidade de mais uma GPU ajuda a diminuir não só o tempo Global, como também o Individual (speedup de 141/88=1,60,

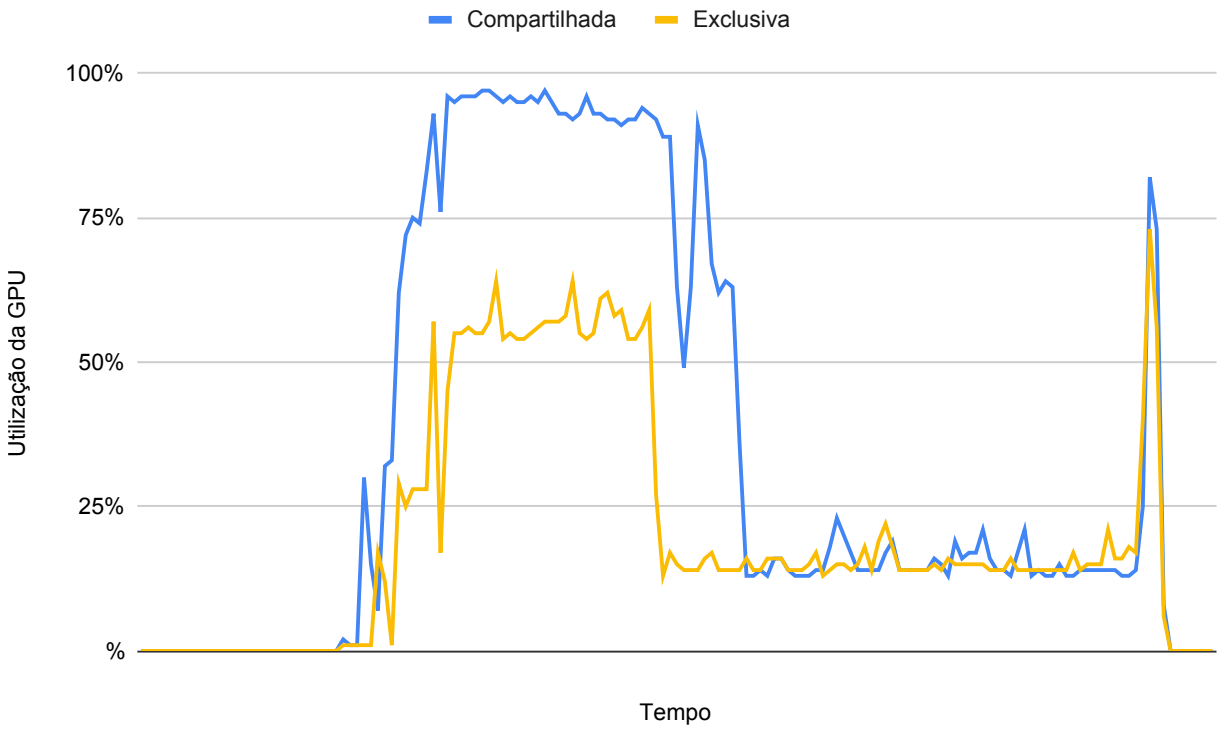


Figura 4.6: Níveis de utilização (carga) de uma GPU em diferentes modos de utilização.

semelhante ao *speedup* dos tempos Globais): com mais GPUs, há mais oportunidades de uma *workload* progredir quando as outras não estiverem a competir pelas mesmas GPUs.

Convém realçar que estas conclusões são inerentes ao ambiente e metodologia de teste utilizada, com GPUs homogéneas e *workloads* homogéneas, podendo alterar-se com modificações nestes dois fatores.

GPUs Remotas Partilhadas

Outro cenário de utilização do HaaS simples, mas relevante, corresponde à exploração de apenas GPUs remotas, em modo partilhado. Os tempos de execução deste cenário são apresentados na Figura 4.7, para o uso de 1 ou 2 GPUs remotas.

Estes resultados mostram uma tendência similar à já observada com o uso de GPUs locais partilhadas (Figura 4.4), em que a adição de uma segunda GPU favorece o desempenho. Todavia, com GPUs remotas o *speedup* em termos Globais, gerado por essa adição, é

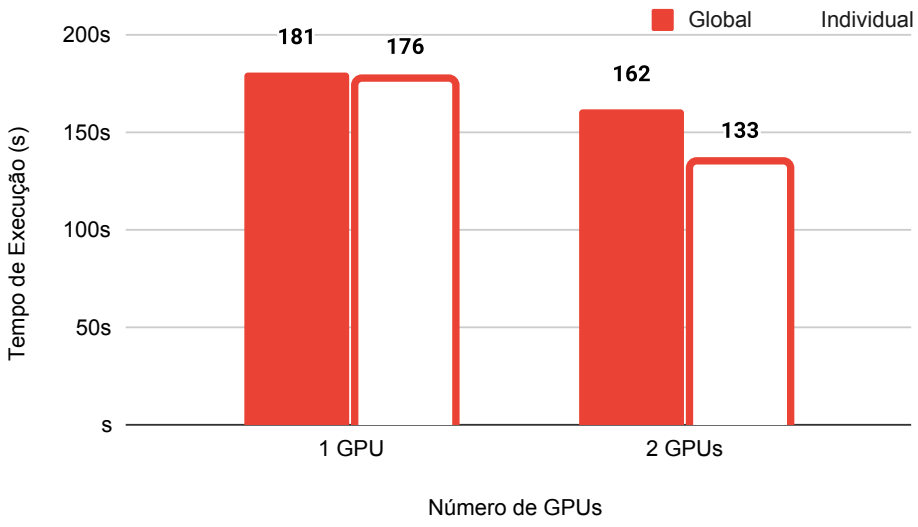


Figura 4.7: Tempos de execução com GPUs remotas partilhadas.

de apenas 181/162 = 1.12, bastante inferior ao observado com GPU locais (1.67). Adicionalmente, é expectável que o uso só de GPUs remotas traduz-se num desempenho inferior ao proporcionado pelas GPUs locais; estas, em comparação com as remotas, oferecem uma aceleração de 181/150=1.21 para 1 GPU, e de 162/123=1.32 para 2 GPUs.

Os tempos Individuais seguem também a tendência da Figura 4.4, sendo ligeiramente inferiores aos Globais; no entanto, devido à natureza remota das GPUs, são superiores[1].

A explicação para um menor desempenho deste cenário, quer em termos absolutos, quer em termos relativos, quando comparado com o da Figura 4.4, reside, obviamente, na penalização imposta pela rede, através da qual circulam todos os pedidos OpenCL gerados pelo *container* Hashcat que corre no primeiro nó do ambiente de testes, e que são redirecionados para o segundo nó. O fato de adição de uma segunda GPU remota produzir um *speedup* anémico (1.12) tem também a ver com o aumento da sobrecarga de comunicação ao nível do rOpenCL, limitando o alcance dos ganhos de desempenho.

[1]Os tempos Individuais serão omissos na análises seguintes, voltando a surgir apenas caso se justifique.

GPUs Locais Partilhadas e Remotas Partilhadas

Mantendo o modo de execução partilhada, o ambiente de testes montado permite ainda avaliar cenários onde se tira partido de mais que 2 GPUs, combinando o uso de 2 GPUs locais, com 1 ou 2 GPUs remotas, num total de 3 ou 4 GPUs utilizadas.

A Figura 4.8 permite observar e comparar a evolução dos tempos de execução Globais das 3 *workloads*, à medida que se aumenta o número de GPUs partilhadas. Nesta figura é introduzida ainda uma notação, para identificar claramente as configurações de GPUs usadas, e que será adotada nos testes seguintes: 1Lp – 1 GPU Local partilhada; 2Lp – 2 GPUs Locais partilhadas; 2Lp+1Rp – 2 GPUs Locais partilhadas num nó rOpenCL, e 1 GPU Remota partilhada noutro nó; 2Lp+2Rp - 2 GPUs Locais partilhadas num nó rOpenCL, e 2 GPUs Remotas partilhadas noutro nó. Os tempos para as configurações 1Lp e 2Lp são os já mostrados na Figura 4.4 para 1 GPU e 2 GPUs, respetivamente.

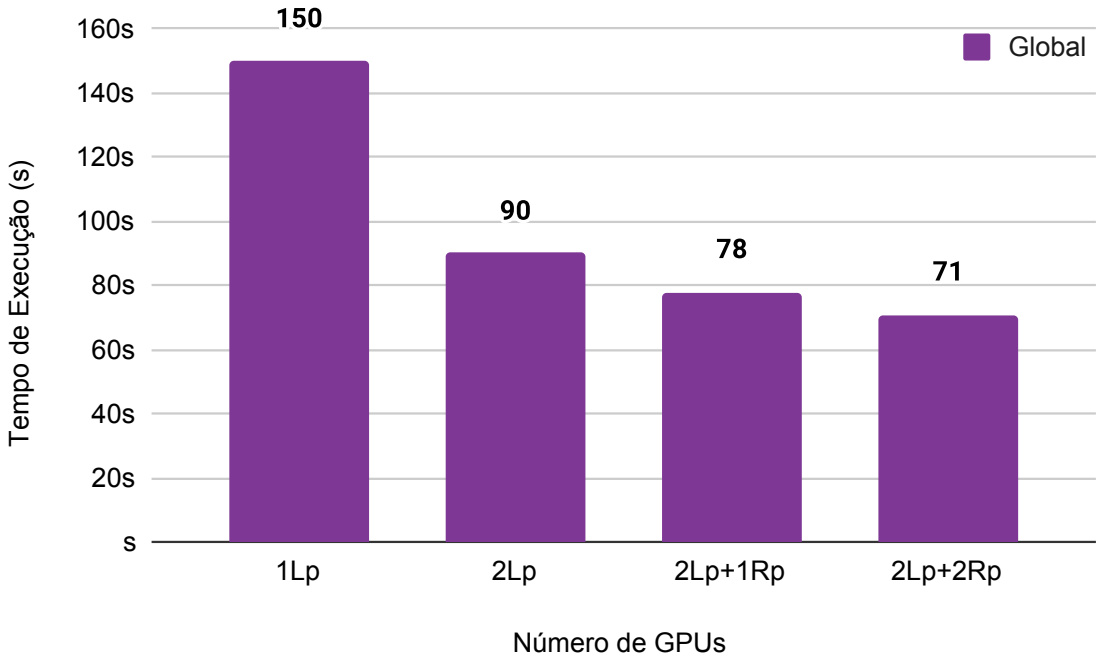


Figura 4.8: Tempos de execução com GPUs partilhadas (locais e remotas).

Neste teste é possível ver que a adição de GPUs remotas aumenta o desempenho, embora não tão significativamente quanto a adição de GPUs locais. Assim, enquanto que a adição da segunda GPU local produz um acréscimo de 67% (150/90), a adição da

primeira GPU remota apenas acrescenta 15% (90/78) e a da segunda GPU remota oferece mais 10% (78/71). Por outro lado, comparando o uso de 1 GPU local, com o uso das 4 GPUs (2 locais e 2 remotas), esta última configuração proporciona uma aceleração de 2.11 (150/71); ou seja, são precisas 4 GPUs para conseguir diminuir o tempo de execução para cerca de metade. Comparando ainda o uso de 2 GPUs locais com as 4 GPUs, o speedup proporcionado por estas é de 90/71=1.27; portanto, o uso de GPUs remotas permitiu ir buscar mais cerca de 1/4 de desempenho, o que não é de todo irrelevante.

4.4.2 1 serviço HaaS, 2 Modos de Execução

Uma outra forma de utilizar a plataforma HaaS é através de cenários híbridos, em que os dois modos de utilização das GPUs são explorados em simultâneo. Nesta secção, foram considerados para avaliação os seguintes cenários desse género:

- **1Le1Lp**: 1 GPU Local exclusiva, 1 GPU Local partilhada;
- **1Le+1Rp**: 1 GPU Local exclusiva, 1 GPU Remota partilhada;
- **2Le+1Rp**: 2 GPUs Locais exclusivas, 1 GPU Remota partilhada;
- **1Le1Lp+1Rp**: 1 GPU Local exclusiva, 1 GPU Local partilhada, 1 GPU Remota partilhada;
- **1Le1Lp+2Rp**: 1 GPU Local exclusiva, 1 GPU Local partilhada, 2 GPUs Remotas partilhada;
- **2Le+2Rp**: 2 GPUs Locais exclusivas, 2 GPUs Remotas partilhadas;

De referir que estes cenários são naturalmente impostos pelo facto de se continuar a ter apenas um serviço HaaS em execução (não havendo portanto suporte a migração), e por restrições da arquitetura e da implementação atual do HaaS (GPUs exclusivas têm de ser locais, e GPUs remotas só podem ser partilhadas). Recorde-se ainda que para avaliar cada um destes cenários, continuam a ser lançadas 3 *workloads* idênticas, que usarão

em simultâneo todas as GPUs partilhadas disponíveis; quanto às exclusivas, se o cenário definir apenas uma dessas GPUs, ela será usada à vez, tirando partido dos mecanismos de reserva; se o cenário definir duas dessas GPUs, cada uma será usada por uma *workload* distinta, e a primeira GPU que ficar livre será atribuída à *workload* em fila de espera (notar, no entanto, que esta *workload* pôde, desde o início, usar as GPUs partilhadas).

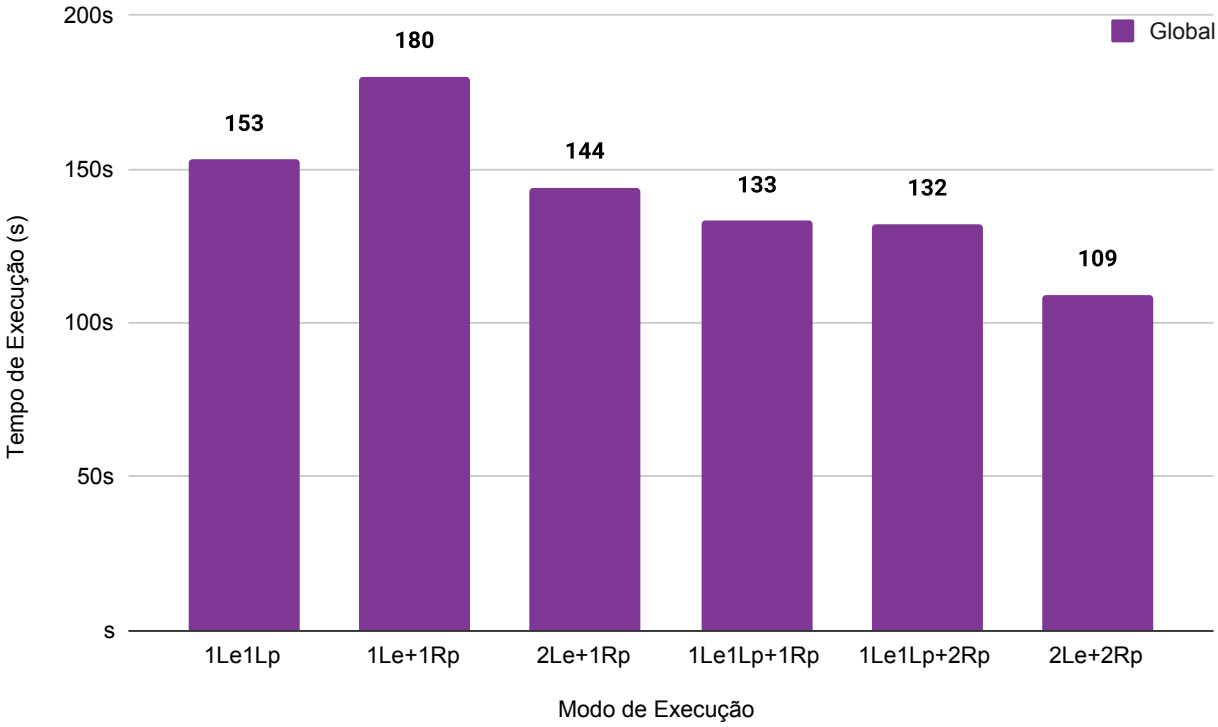


Figura 4.9: Tempo de execução dos cenários híbridos.

Os tempos de execução para os cenários híbridos constam da Figura 4.9. Analisando esta figura, podem-se retirar algumas conclusões:

- Nenhum destes cenários produz um resultado melhor que o obtido logo com 2 GPUs partilhadas locais (90s – rever Figura 4.8); de facto, nos cenários híbridos é obrigatório que pelo menos uma das GPUs locais seja exclusiva e, como já se viu anteriormente (Figura 4.5), isso prejudica o tempo Global de execução (embora favoreça o Individual), e nem a adição de GPUs remotas partilhadas é suficiente compensador;
- O cenário 1Le+1Rp (180s) é pior que o 1Le1Lp (153s), devido à substituição de uma GPU local partilhada, por uma remota partilhada

- O cenários 2Le+1Rp (144s) é melhor que o 1Le+1Rp (180s), devido à adição de uma GPU local exclusiva, passando a ter mais um GPU; o aumento do número de GPUs utilizadas foi também suficiente para melhorar face ao cenário de ponto de partida 1Le1Lp (153s);
- O cenário 1Le1Lp+1Rp (133s) melhora face ao 2Le+1Rp (144s), porque uma das GPUs locais exclusivas passou a partilhada;
- No cenário 1Le1Lp+2Rp (132s), a adição de mais uma GPU partilhada quase que não melhora o desempenho (132), dado a GPU adicional ser remota
- O cenário 2Le+2Rp (109s) usa 4 GPUs, tal como o cenário 1Le1Lp+2Rp (132s), mas é notoriamente mais rápido: 2 das workloads usam uma GPU exclusiva cada, sem qualquer competição; o cenário 2Le+2Rp (109s) é também diretamente comparável com o 2Le+1Rp (144s), ao qual acrescenta uma GPU remota partilhada, sendo por isso mais rápido, devido ao acréscimo do nº de GPUs
- À medida que sobe o nº de GPUs usadas, o desempenho melhora; porém, devido à restrição de pelo menos uma GPUs local ser exclusiva, não se conseguem alcançar os níveis de desempenho oferecidos por 2 GPUs locais partilhadas (rever Figura 4.8);
- Nestes cenários, os que oferecem melhor desempenho usam 2 GPUs locais; isso sugere que uma *workload* pode beneficiar tirando partido da migração para maximizar o número de GPUs locais utilizadas, ou conseguir utilizar GPUs locais mais rápidas.

4.4.3 2 serviços HaaS

Outra estratégia de exploração suportada pelo HaaS é a colocação de instâncias do Hashcat em diferentes máquinas, com possibilidade de migração. Nalguns casos, isso poderá permitir extrair maior desempenho das GPUs disponíveis.

Para começar, a dispersão permite definir em modo exclusivo todas as GPUs do *cluster*, o que implica arrancar as instâncias Hashcat que as selecionaram, nas máquinas que as albergam, dado que esse tipo de GPUs tem de ser local sob o ponto de vista das instâncias.

A migração permite que, durante a sua execução, uma instância possa alterar a GPU exclusiva (ou adicionar uma, caso não a tenha), o que pode implicar a movimentação da instância para outro nó do *cluster* (o nó que alberga essa GPU), tudo isto sem perder acesso às GPUs partilhadas (GPUs partilhadas que eram locais passam a remotas, e vice-versa, sendo que o rOpenCL continua a permitir o acesso transparente a todas).

Para avaliar o impacto dos mecanismos de migração, foram realizados vários testes em diferentes cenários. Nestes cenários continuam a ser executadas 3 *workloads* (instâncias Hashcat) homogéneas, criadas em simultâneo. Todas as *workloads* solicitam uma GPU exclusiva; se esta estiver disponível, ou se houver GPUs partilhadas, arrancam de imediato; se a GPU exclusiva ficar disponível mais tarde, poderá haver migração da *workload*, dependendo do nó onde foi originalmente criada. Esses cenários enumeram-se a seguir, com base numa notação semelhante à usada anteriormente, mas onde a localidade (L vs R) das GPUs é omitida, sendo apenas relevante em que máquina/nó estão situadas:

- **1e+1e**: 2 GPUs exclusivas, 1 em cada nó; as *workloads* 1 e 2 arrancam cada uma num nó diferente (devido ao requisito da GPU exclusiva ser local); a *workload* 3 só arranca quando uma GPU exclusiva ficar livre, sendo criada no nó respetivo; neste caso não existe, portanto, migração, apenas se usando o mecanismo de reserva;

- **1e+1p**: 1 GPU exclusiva e 1 partilhada, 1 em cada nó; a *workload* 1 é criada no nó 1, de forma a usar a GPU exclusiva desse nó; as *workloads* 2 e 3 podem ser criadas em qualquer nó, cabendo ao Docker Swarm essa decisão (pelo menos uma será criada no nó 2, dado que o nó 1 terá já a *workload* 1); todas as *workloads* partilham a GPU do nó 2 (que será remota para a *workload* 1, e poderá ser local ou remota para as outras *workloads*, dependendo em que nó foram arrancadas); quando a *workload* 1 terminar, uma das outras irá usar a GPU exclusiva, migrando para o nó 1 se for preciso (e o mesmo acontecerá com a *workload* remanescente);

- **1e+2p**: 1 GPU exclusiva e 2 partilhadas, a exclusiva em um nó, e as partilhadas noutro; este cenário á parecido com o anterior (**1e+1p**), acrescentando-lhe apenas uma GPU partilhada, o que deverá melhorar o desempenho;

- **2e+1p**: 2 GPUs exclusivas e 1 partilhada, as exclusivas em um nó, e a partilhada noutro; as *workloads* 1 e 2 são criadas no nó 1, cada uma usando uma GPU exclusiva desse nó; a *workload* 3 é muito provavelmente criada no nó 2; todas as *workloads* partilham a GPU do nó 2; quando uma das *workloads* 1 ou 2 terminar, caso a *workload* 3 ainda não tenha terminado, terá acesso a uma GPU exclusiva, migrando para o nó 1 se necessário; este cenário deverá ser mais rápido que o cenário **1e+1p**;

- **1e1p+1e1p**: 2 GPUs exclusivas e 2 partilhadas, cada nó com 1 exclusiva e 1 partilhada; a *workload* 1 é criada no nó 1, e a *workload* 2 é criada no nó 2; a *workload* 3 será criada num dos dois nós; todas as *workloads* irão usar as 2 GPUs partilhadas; logo que termine uma das *workloads* 1 ou 2, a *workload* 3, caso não tenha terminado, terá acesso a uma GPU exclusiva, migrando para o respetivo nó, se for necessário; este cenário é previsivelmente mais rápido que todos os anteriores, dado que as GPUs exclusivas estão em máquinas distintas, e todas as outras GPUs disponíveis são ainda usadas (em modo partilhado).

- **2e+1e**: 3 GPUs exclusivas, 2 em um nó e 1 noutro; duas *workloads* são criadas no nó 1, e uma é criada no nó 2, tendo cada uma acesso a uma GPU exclusiva; todas as *workloads* executam simultaneamente sem qualquer competição por GPUs (não ha GPUs partilhadas e não há necessidade de migração), pelo que é expectável que este cenário seja mais rápido que todos os anteriores;

- **2e+1e1p**: 3 GPUs exclusivas e 1 partilhada, 2 exclusivas em um nó, e 1 exclusiva e 1 partilhada noutro; este cenário é parecido com o anterior (**2e+1e**), ao qual acrescenta uma GPU partilhada, o que deverá permitir melhorar o desempenho;

O resultado da avaliação destes cenários é apresentado na Figura 4.10, tendo-se comprovado as expetativas relacionados com o desempenho esperado:

- O cenário **1e+1e** é comparável com o teste já realizado com 2 GPUs exclusivas na mesma máquina (ver Figura 4.5, secção 4.4.1), apresentando um tempo ligeiramente melhor (119s vs 123s), dado que agora foram usadas máquinas diferentes.

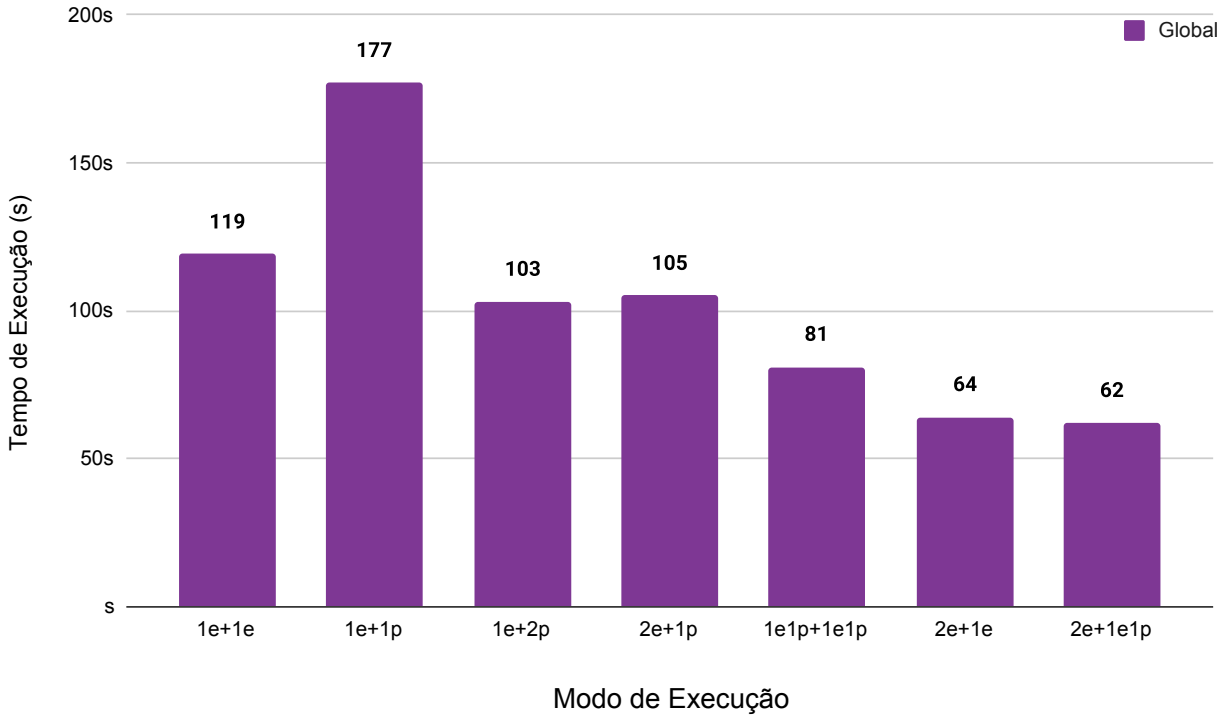


Figura 4.10: Tempo de execução com 2 nós.

- O cenário **1e+1p** (177s) é comparável com o **1Le+1Rp** (180s) (ver Figura 4.9 da secção anterior), notando-se de novo o efeito positivo do uso de 2 máquinas; e o mesmo se passa comparando o cenário **2e+1p** (105s) com o **2Le+1Rp** (144s) (ver mesma figura), sendo neste caso mais pronunciado o benefício das duas máquinas.

- Confirma-se que o cenário **1e1p+1e1p** é mais rápido que todos os anteriores testados nesta secção; e verifica-se que os cenários **2e+1e** e **2e+1e1p** melhoram, sucessivamente, o desempenho, como se tinha previsto, sendo ambos mais rápidos que qualquer dos cenários avaliados nos testes das secções anteriores (em particular, superam o melhor tempo obtido anteriormente, no cenário **2Lp+2Rp**, com todas as GPUs partilhadas - ver Figura 4.8); notar ainda que não faz sentido uma comparação direta dos cenários **1e1p+1e1p**, **2e+1e** e **2e+1e1p**, com os da Figura 4.9, porque nos cenários híbridos não são admissíveis GPU remotas exclusivas.

- O cenário **2e+1e1p** produziu o melhor desempenho de todos os testes realizados.

- Para cenários similares entre esta secção e a anterior, em termos de GPUs partilhadas vs exclusivas, e locais vs remotas, a adição de um nó melhora o desempenho. Isto indicia que a adição de mais nós e de mais GPUs irá melhorar ainda mais o desempenho, mas testes com essas configurações ficaram para trabalho futuro.

4.5 Discussão

Os testes realizados permitiram validar as decisões arquiteturais tomadas, evidenciando o seu impacto. Tornaram também evidente a capacidade do HaaS em se adaptar (nalguns casos dinamicamente) às necessidades específicas de cada cenário de uso.

Os resultados dos testes apresentados, desde os cenários mais simples aos mais complexos, demonstraram consistência e foram de encontro às expetativas. Demonstraram ainda a robustez do sistema ao lidar com diferentes configurações e cargas de trabalho.

Os testes também revelaram algumas limitações na utilização de GPUs remotas, devido ao atraso introduzido pela rede (algo intrínseco à plataforma rOpenCL, conforme avaliado em [3]). Ainda assim a utilização de GPUs remotas é uma potencial mais valia no desempenho do HaaS. De forma semelhante, a divisão dos testes entre dois nós (mantendo o mesmo conjunto de GPUs), demonstrou potencial melhoria no desempenho.

De referir ainda a não existência de uma configuração ideal para todas as situações de uso do HaaS. A fim de maximizar a eficiência e o desempenho, é necessário considerar fatores como a quantidade de nós, GPUs, e cargas de trabalho, bem como a suas características em termos de desempenho oferecido e exigido. No caso das GPUs, é também preciso considerar a sua localidade, e o modo de uso adotado (partilhado vs exclusivo). Por exemplo, em cenários com um único nó, o melhor desempenho encontrado foi com todas as GPUs em modo partilhado. Em contrapartida, ao utilizar dois nós, ter todas as GPUs em modo exclusivo garante melhores resultados, que ainda beneficiam da adição de GPUs partilhadas (caso estejam disponíveis). Todavia, estas conclusões são aplicáveis para o tipo de *hash* testado, podendo eventualmente mudar para outros tipos de *hash*.

Os testes ainda evidenciaram pontos de otimização e/ou melhorias durante sua execução e/ou análise, alguns dos quais são referidos em trabalho futuro, no capítulo a seguir.

Capítulo 5

Conclusões

Este trabalho explorou o desenvolvimento de uma plataforma inovadora para a recuperação de senhas, abordando os desafios crescentes na área de segurança cibernética. Ao longo do trabalho, investigou-se a complexidade da quebra de senhas, a compreensão dos algoritmos de criptografia e ainda a exploração de tecnologias para aumentar o desempenho das ferramentas conhecidas, como GPUs e computação em *cluster*.

Foi desenvolvido o HaaS, uma plataforma que oferece o Hashcat como um serviço, aproveitando o poder de GPUs locais e remotas para acelerar o processo de quebra de senhas. Para isso foi necessário a criação de serviços para gerenciamento de usuários, filas de espera, *checkpointing* de estado, e suporte a migração de *containers* entre nós, visando otimizar o desempenho e a eficiência do sistema.

A avaliação do HaaS demonstrou resultados promissores, confirmando a viabilidade técnica da abordagem e a sua capacidade de lidar com cargas de trabalho intensivas. Observou-se que a escolha da configuração ideal do sistema depende de vários fatores, como o número de cargas de trabalho (utilizadores em simultâneo) e de GPUs disponíveis.

No seu estágio atual, o HaaS permite a ampla utilização de suas funcionalidades de forma consistente e robusta, com base no código disponível em [68]. As imagens Docker utilizadas, com as versões mais atuais de cada serviços, são públicas e podem ser descarregadas a partir do Docker Hub: carlossouzal/hashcloud-main [75], carlossouzal/hashcat [76], carlossouzal/hashcloud-users [77] e carlossouzal/hashcloud-client [78].

Em suma, este trabalho contribuiu para o avanço do conhecimento na área de recuperação de senhas, fornecendo uma plataforma prática e eficiente para lidar com os desafios crescentes da segurança cibernética. O HaaS, com sua arquitetura inovadora e recursos, tem o potencial de se tornar uma ferramenta valiosa para profissionais de segurança e pesquisadores, auxiliando na proteção de dados e sistemas contra ameaças cibernéticas.

5.1 Trabalhos Futuros

Pode-se portanto afirmar que os objetivos iniciais desta dissertação foram alcançados. Porém, durante o processo de desenvolvimento, novas oportunidades de melhoria puderam ser identificadas. Algumas destas, remetidas para trabalho futuro, são:

- Suportar a reserva de GPUs em modo partilhado, a fim de de evitar a sua sobrecarga.
- Tirar proveito de mais tipos de co-processadores suportados pelo rOpenCL, além de GPUs, pois o Haschcat é também capaz de usar CPUs.
- Aumentar o número de dispositivos exclusivos permitidos por cada instância (atualmente, cada *container* Hashcat pode usar um só acelerados exclusivo).
- Aprimorar o gerenciamento de dispositivos, trocando o gerenciamento de recursos individuais, por tipos de recursos (escolha baseada em modelo, desempenho, etc.).
- Realizar a avaliação da plataforma HaaS com um maior número de GPUs e nós, e testar outros algoritmos de *hash* suportados pelo Hashcat.
- Realizar a avaliação da plataforma HaaS com diferentes configurações de rede (MTU, velocidade, etc.), dado a sua influência no desempenho do rOpenCL.
- Partilhar este trabalho com a comunidade científica através de publicações.

Bibliografia

[1] hashcat, *Hashcat: Advanced Password Recovery*, versão 6.2.5, Acesso em: 09/10/2023, 2022. URL: https://hashcat.net/hashcat/.

[2] R. Hranický, L. Zobal, V. Večeřa e P. Matousek, «Distributed Password Cracking in a Hybrid Environment,» mai. de 2017.

[3] R. Alves, «rOpenCL: uma ferramenta para acesso de aplicações heterogéneas a co-processadores remotos,» por, Accepted: 2021-02-04T15:54:50Z, tese de mestrado, 2020. URL: https://bibliotecadigital.ipb.pt/handle/10198/23222 (acedido em 22/10/2023).

[4] S. Gold, «Cracking passwords,» *Network Security*, vol. 2010, n.º 8, pp. 4–7, 2010, ISSN: 1353-4858. DOI: https://doi.org/10.1016/S1353-4858(10)70103-3. URL: https://www.sciencedirect.com/science/article/pii/S1353485810701033.

[5] B. Preneel, «Cryptographic hash functions,» *European Transactions on Telecommunications*, vol. 5, n.º 4, pp. 431–448, 1994.

[6] E. Thompson, «MD5 collisions and the impact on computer forensics,» *Digital Investigation*, vol. 2, n.º 1, pp. 36–40, 2005, ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2005.01.004. URL: https://www.sciencedirect.com/science/article/pii/S1742287605000058.

[7] H. Gilbert e H. Handschuh, «Security Analysis of SHA-256 and Sisters,» em *Selected Areas in Cryptography*, M. Matsui e R. J. Zuccherato, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 175–193, ISBN: 978-3-540-24654-1.

[8] N. Provos e D. Mazieres, «Bcrypt algorithm,» em *USENIX*, 1999.

[9] C. Percival e S. Josefsson, «The scrypt password-based key derivation function,» rel. téc., 2016.

[10] A. Biryukov, D. Dinu e D. Khovratovich, «Argon2: new generation of memory-hard functions for password hashing and other applications,» em *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2016, pp. 292–302.

[11] L. R. Knudsen e M. J. B. Robshaw, «Brute Force Attacks,» em *The Block Cipher Companion*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 95–108, ISBN: 978-3-642-17342-4. DOI: 10.1007/978-3-642-17342-4_5. URL: https://doi.org/10.1007/978-3-642-17342-4_5.

[12] A. Narayanan e V. Shmatikov, «Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff,» em *Proceedings of the 12th ACM Conference on Computer and Communications Security*, sér. CCS '05, Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 364–372, ISBN: 1595932267. DOI: 10.1145/1102120.1102168. URL: https://doi.org/10.1145/1102120.1102168.

[13] H. Kumar, S. Kumar, R. Joseph et al., «Rainbow table to crack password using MD5 hashing algorithm,» em *2013 IEEE Conference on Information & Communication Technologies*, 2013, pp. 433–439. DOI: 10.1109/CICT.2013.6558135.

[14] F. Salahdine e N. Kaabouch, «Social Engineering Attacks: A Survey,» *Future Internet*, vol. 11, n.º 4, 2019, ISSN: 1999-5903. DOI: 10.3390/fi11040089. URL: https://www.mdpi.com/1999-5903/11/4/89.

[15] L. Casati e A. Visconti, «Exploiting a Bad User Practice to Retrieve Data Leakage on Android Password Managers,» em *Innovative Mobile and Internet Services in Ubiquitous Computing*, L. Barolli e T. Enokido, eds., Cham: Springer International Publishing, 2018, pp. 952–958, ISBN: 978-3-319-61542-4.

[16] A. Narayanan e V. Shmatikov, «Fast Dictionary Attacks on Passwords Using Time-Space Tradeoff,» em *Proceedings of the 12th ACM Conference on Computer and Communications Security*, sér. CCS '05, Alexandria, VA, USA: Association for Computing Machinery, 2005, pp. 364–372, ISBN: 1595932267. DOI: 10.1145/1102120.1102168. URL: https://doi.org/10.1145/1102120.1102168.

[17] M. Weir, S. Aggarwal, B. de Medeiros e B. Glodek, «Password Cracking Using Probabilistic Context-Free Grammars,» mai. de 2009, pp. 391–405. DOI: 10.1109/SP.2009.8.

[18] M. A. Fauzi, B. Yang e E. Martiri, «PassGAN for Honeywords: Evaluating the Defender and the Attacker Strategies,» em *Advances on Smart and Soft Computing*, F. Saeed, T. Al-Hadhrami, F. Mohammed e E. Mohammed, eds., Singapore: Springer Singapore, 2021, pp. 391–401, ISBN: 978-981-15-6048-4.

[19] E. Zhou, Y. Peng, G. Shao, F. Deng, Y. Miao e W. Fan, «Password cracking using chunk similarity,» *Future Generation Computer Systems*, vol. 150, pp. 380–394, 2024, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.09.013. URL: https://www.sciencedirect.com/science/article/pii/S0167739X23003382.

[20] *Cain & Abel*, Acesso em: 09/10/2023. URL: http://www.cs.toronto.edu/~arnold/427/15s/csc427/tools/CainAndAbel/index.html.

[21] *Cain & Abel*, https://github.com/xchwarze/Cain, Acesso em: 09/10/2023, 2014. URL: https://github.com/xchwarze/Cain.

[22] *RainbowCrack*, http://project-rainbowcrack.com/, versão 1.8, Acesso em: 13/07/2023, 2020. URL: http://project-rainbowcrack.com/.

[23] Y. Tabata, K. Iwai, H. Tanaka e T. Kurokawa, «Improved GPU Implementation of RainbowCrack,» em *2015 Third International Symposium on Computing and Networking (CANDAR)*, 2015, pp. 616–618. DOI: 10.1109/CANDAR.2015.32.

[24] *Ophcrack*, https://ophcrack.sourceforge.io, versão 3.8.0, Acesso em: 10/10/2023. URL: https://ophcrack.sourceforge.io.

[25] *Aircrack-ng*, https://www.aircrack-ng.org, versão 1.7, Acesso em: 10/10/2023, 2022. URL: https://www.aircrack-ng.org.

[26] *THC-Hydra*, https://github.com/vanhauser-thc/thc-hydra, versão 9.5, Acesso em: 10/10/2023, 2022. URL: https://github.com/vanhauser-thc/thc-hydra.

[27] *NCrack*, https://nmap.org/ncrack/, versão 7.94, Acesso em: 10/10/2023, 2023. URL: https://nmap.org/ncrack/.

[28] J. the Ripper Project, *John the Ripper: Password Cracker*, versão 1.9.0-jumbo, Acesso em: 09/10/2023, 2019. URL: https://www.openwall.com/john/.

[29] A. W. Services. «Amazon Web Services.» Acesso em: 10/10/2023, Amazon Web Services, Inc. (2023), URL: https://aws.amazon.com/pt/.

[30] *Licença MIT*, https://opensource.org/licenses/MIT, Acesso em: 09/10/2023, 2022.

[31] J. Cheng, M. Grossman e T. McKercher, *Professional CUDA c programming*. John Wiley & Sons, 2014.

[32] A. Munshi, «The opencl specification,» em *2009 IEEE Hot Chips 21 Symposium (HCS)*, IEEE, 2009, pp. 1–314.

[33] P. Rogers e A. Fellow, «Heterogeneous system architecture overview.,» em *Hot Chips Symposium*, 2013, pp. 1–41.

[34] T. Murakami, R. Kasahara e T. Saito, «An implementation and its evaluation of password cracking tool parallelized on GPGPU,» en, em *2010 10th International Symposium on Communications and Information Technologies*, Tokyo, Japan: IEEE, out. de 2010, pp. 534–538, ISBN: 978-1-4244-7007-5. DOI: 10.1109/ISCIT.2010.5665047. URL: http://ieeexplore.ieee.org/document/5665047/ (acedido em 06/10/2023).

[35] M. Sprengers, «GPU-based password cracking,» *On the Security of Password Hashing Schemes regarding Advances in Graphics Processing Units, Radboud University Nijmegen*, 2011.

[36] T. Murakami, R. Kasahara e T. Saito, «An implementation and its evaluation of password cracking tool parallelized on GPGPU,» em *2010 10th International Symposium on Communications and Information Technologies*, IEEE, 2010, pp. 534–538.

[37] *GeForce Graphics Cards - Ultimate PC Gaming*, en-eu. URL: https://www.nvidia.com/en-eu/geforce/ (acedido em 22/10/2023).

[38] *The Khronos Group*, en, Section: General, out. de 2023. URL: https://www.khronos.org (acedido em 22/10/2023).

[39] M. Shimizu e A. Yonezawa, «Remote process execution and remote file I/O for heterogeneous processors in cluster systems,» em *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, IEEE, 2010, pp. 145–154.

[40] R. Alves e J. Rufino, «Extending Heterogeneous Applications to Remote Co-processors with rOpenCL,» set. de 2020, pp. 305–312. DOI: 10.1109/SBAC-PAD49847.2020.00049.

[41] A. Zonenberg, «Distributed hash cracker: A cross-platform gpu-accelerated password recovery system,» mai. de 2009.

[42] D. Apostal, K. Foerster, A. Chatterjee e T. Desell, «Password recovery using MPI and CUDA,» em *2012 19th International Conference on High Performance Computing*, 2012, pp. 1–9. DOI: 10.1109/HiPC.2012.6507505.

[43] P. Kamal, «A Study on the Security of Password Hashing Based on GPU Based, Password Cracking using High-Performance Cloud Computing,» en,

[44] R. Hranický, M. Holkovič e P. Matoušek, «On Efficiency of Distributed Password Recovery,» en, *Journal of Digital Forensics, Security and Law*, 2016, ISSN: 15587223. DOI: 10.15394/jdfsl.2016.1380. URL: http://commons.erau.edu/jdfsl/vol11/iss2/5/ (acedido em 06/10/2023).

[45] R. Hranický, L. Zobal, O. Ryšavý e D. Kolář, «Distributed password cracking with BOINC and hashcat,» en, *Digital Investigation*, vol. 30, pp. 161–172, set. de 2019, ISSN: 17422876. DOI: 10.1016/j.diin.2019.08.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S1742287619301446 (acedido em 06/10/2023).

[46] *hashtopolis/server: Hashtopolis - A Hashcat wrapper for distributed password recovery*. URL: https://github.com/hashtopolis/server (acedido em 22/10/2023).

[47] M. H. Syed, E. B. Fernandez et al., «The software container pattern,» em *Proceedings of the 22nd Conference on Pattern Languages of Programs*, 2015, pp. 24–26.

[48] I. Docker, *Docker - Containerization Platform*, https://www.docker.com/, Acessado em 22 de outubro de 2023, 2013.

[49] P. Authors, *Podman: A tool for managing OCI containers and pods*, Acessado em 22 de outubro de 2023, 2018.

[50] rkt Authors, *rkt - the pod-native container engine*, Acessado em 22 de outubro de 2023, 2014.

[51] I. Docker. «Docker Swarm - Native Clustering and Orchestration for Docker.» Acessado em 19 de outubro de 2023. (2019), URL: https://docs.docker.com/engine/swarm/.

[52] T. K. Authors, *Kubernetes: Production-Grade Container Orchestration*, v1.28, Acessado em 22 de outubro de 2023, Cloud Native Computing Foundation, 2014.

[53] I. Docker, *Docker Compose overview*, 2023. URL: https://docs.docker.com/compose/.

[54] S. Shepler, B. Callaghan, D. Robinson et al., «Network file system (NFS) version 4 protocol,» rel. téc., 2003.

[55] MongoDB, Inc., *MongoDB Manual*, https://www.mongodb.com/docs/manual/, MongoDB Manual v7.0, 2024.

[56] Redis Labs, *Redis Documentation*, Acessado em 8 de junho de 2024, 2024. URL: https://redis.io/docs/latest/.

[57] C. N. C. Foundation, *Keycloak*, 2023. URL: https://www.keycloak.org/.

[58] I. A. Mohammed, «Systematic review of Identity Access Management in information security,» *International Journal of Innovations in Engineering Research and Technology*, vol. 4, n.º 7, pp. 1–7, 2017.

[59] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[60] J. Thönes, «Microservices,» *IEEE Software*, vol. 32, n.º 1, pp. 116–116, 2015. DOI: 10.1109/MS.2015.11.

[61] K. Inc., «Kong Gateway: The Worldś Most Adopted Open Source API Gateway,» 2023. URL: https://konghq.com/products/kong-gateway.

[62] The Go Authors, *Go Documentation*, Acesso em 15 de junho de 2024, 2023. URL: https://go.dev/doc/.

[63] Microsoft Corporation, *TypeScript*, Acesso em 15 de junho de 2024, 2023. URL: https://www.typescriptlang.org/.

[64] M. Contributors, *JavaScript | MDN*, 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[65] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[66] P. Smith, *analyze hc restore*, https://github.com/philsmd/analyze_hc_restore, 2023.

[67] NVIDIA, *NVIDIA Container Runtime on NVIDIA Developer*, 2023. URL: https://developer.nvidia.com/container-runtime.

[68] C. Lima, *Hashcat as a Service*, https://gitlab.estig.ipb.pt/ipb1/estig/mi/thesis/2022_2023/hashcatasservice/code/-/tree/main/server, Acesso em 29 de maio de 2024, 2023.

[69] I. Docker, *docker service*, Docker, 2023. URL: https://docs.docker.com/reference/cli/docker/service/.

[70] I. Docker, *docker container*, Docker, 2023. URL: https://docs.docker.com/reference/cli/docker/container/.

[71] Vercel, *Next.js*, https://nextjs.org, Acesso em 29 de maio de 2024, 2023.

[72] Redux maintainers, *Redux*, https://redux.js.org/, 2024.

[73] H. Dobbertin, «Cryptanalysis of MD5 compress,» *rump session of Eurocrypt*, vol. 96, pp. 71–82, 1996.

[74] D. Rachmawati, J. Tarigan e A. Ginting, «A comparative study of Message Digest 5 (MD5) and SHA256 algorithm,» em *Journal of Physics: Conference Series*, IOP Publishing, vol. 978, 2018, p. 012 116.

[75] C. Lima, *hashcloud-main*, https://hub.docker.com/repository/docker/carlossouzal/hashcloud-main/general, Acessado em 30 de maio de 2024.

[76] C. Lima, *hashcat*, https://hub.docker.com/repository/docker/carlossouzal/hashcat/general, Acessado em 30 de maio de 2024.

[77] C. Lima, *hashcloud-users*, https://hub.docker.com/repository/docker/carlossouzal/hashcloud-users/general, Acessado em 30 de maio de 2024.

[78] C. Lima, *hashcloud-client*, https://hub.docker.com/repository/docker/carlossouzal/hashcloud-client/general, Acessado em 30 de maio de 2024.

Apêndice A

Proposta Original do Projeto

INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

MESTRADO EM INFORMÁTICA

Master of Informatics

Unidade Curricular de "Dissertação/Projeto/Estágio"

Course unit "Thesis/Project/Intership"

Work proposal

Proposta de tema

☑ Dissertação *Thesis* ☐ Projeto *Project* ☐ Estágio *Internship*

Título *Title*

HaaS - Hashcat como Serviço

Palavras-chave *Keywords*

Desenvolvimento Web, Criptografia Aplicada, Computação Paralela e Distribuída, Virtualização e Containerização

Orientador IPB *IPB Supervisor*	*email*
José Carlos Rufino Amaro	rufino@ipb.pt

Co-orientador IPB *IPB Co-supervisor*	*email*

Co-orientador externo 1 *External co-supervisor 1*	*email*
André Koscianski	koscianski@utfpr.edu.br

Instituição do Co-orientador externo 1 *Institution of external co-supervisor 1*	País *Country*
UTFPR	Brasil

Co-Orientador externo 2 *External co-supervisor 2*	*email*

Instituição do Co-orientador externo 2 *Institution of external co-supervisor 2*	*Country*

Aluno *Student*

n º.	Nome *name*	email *email*
54190	**Carlos de Souza Lima**	a54190@alunos.ipb.pt

Objetivos *Goals*

Desenhar, implementar e validar uma plataforma que permita lançar, via interface web, instâncias do utilitário Hashcat para execução em containers, com suporte à migração entre nós de computação e exploração de aceleradores locais e remotos.

Descrição *Description*

No mundo da criptografia e cibersegurança, o Hashcat é um ferramenta muito popular. De forma resumida, o Hashcat possibilita o uso de placas gráficas (GPUs) para acelerar a recuperação de passwords com base nos seus hashes criptográficos. A rapidez do Hashcat está diretamente ligada com o tipo da cifra usada para originalmente encriptar a password, e com o número e capacidade computacional das GPUs usadas para a recuperar. Uma vez que o número de GPUs que um só computador pode alojar é necessariamente limitado, uma solução que permita ao Hashcat usar GPUs remotas introduz um fator de aceleração importante para o processo de recuperação da password.

Nesta dissertação, pretende-se implementar uma plataforma que, com base num frontend web, comece por permitir acionar e gerir a execução do Hashcat usando as GPUs locais instaladas na mesma máquina onde o frontend web é executado. Após esta prova de conceito, serão usados containers para fornecer isolamento entre as diferentes instâncias locais de Haschat (de diferentes utilizadores / clientes web). Depois, a plataforma "Hashcat as a Service" será enriquecida com a integração de midleware rOpenCL (remote OpenCL), permitindo assim a utilização de GPUs remotas. Finalmente, a plataforma será dotada da capacidade de migração de containers para outros nós computacionais, tirando partido das capacidades de checkpointing do Hashcat.

Metodologia / Plano de trabalhos *Methodology/ Work plan*

a) Estudo e experimentação com hashcat e GPUs
b) Desenvolvimento do serviço web de prova de conceito
c) Adição de isolamento por meio de containers
d) Estudo e implantação do middleware rOpenCL
e) Integração do rOpenCL com os containers
f) Implementação da mobilidade de containers

Pré-requisitos *Prerequisites*

Conhecimentos de Desenvolvimento Web, Linguagem C/C++/Python, Ambiente de Linha de Comando em Linux

Recursos necessários *Resources needed*

Computador pessoal para desenvolvimento;
Cluster do CeDRI para instanciação, validação e avaliação da plataforma desenvolvida.

Data *Date*

Apêndice B

Manual do utilizador

HaaS - Hashcat as a Service

Hashcat as a Service is a platform designed to create and manage containers built with Hashcat on demand.

Installation

Requirements

Before proceed with the installation make sura that you have already installed

- Docker
- OpenCL
- rOpenCL (see, https://github.com/ruialves7/rOpenCL)

Docker Configuration

Make sure you have Docker Swarm running by running the following command

```
docker swarm init --advertise-addr {YOUR_IP}
```

After that you will be able to see your cluster running

```
docker node ls
```

You will probably only have one node, to add more nodes you can run

```
docker swarm join \
    --token  {TOKEN} \
    {YOUR_IP}:{PORT}
```

If you don't know that TOKEN and PORT need to join, you can use the command

```
docker swarm join-token manager
```

It will return the commands required to join, including TOKEN, IP and PORT.

GPUs labels

Yet, you need to included as label for each node the GPUs reference (UUID_KHR). This information will be used for escalonate correctly and eficiently the Hashcat containers.

You are able to see the UUID_KHR for each GPU using the command

```
clinfo --json
```

The command required to create the label is:

```
docker node update --label-add {UUID_KHR}=exclusive node-{ID}
```

Optionally, you can set just the GPUs that you'll use in exclusive mode.

rOpenCL

On each node it is also important to have the rOpenCL Deamon running.

The steps to achieve this can be seen at https://github.com/ruialves7/rOpenCL/tree/master/installation/rDaemon.

Optionally:

You can also start the rOpenCL Daemon using Systemctl (see https://www.digitalocean.com/community/tutorials/how-to-use-systemctl-to-manage-systemd-services-and-units-pt), or by starting directly the application.

Now, finally, to start the application, you need to go to the *code/server/deploy* folder and execute the following commands:

```
sudo docker network create -d overlay hashcloud-network
sudo docker stack deploy -c docker-compose.kong.yml kong
sudo docker compose -f docker-compose.keycloak.yml up -d
```

Here you need to create a realm and a client in keycloak to be used, see https://medium.com/@dddogaaa/get-started-with-keycloak-8a1f0ad98f2e and https://help.zerto.com/bundle/Linux .ZVM.HTML.10.0_U1/page/Creating_Keycloak_Credentials.htm.

```
sudo docker compose -f docker-compose.redis.yml up -d
```

Update the environment variables contained in docker-compose.app.yml to reflect the previously created services, now run the following command, which will download the images and launch the application.

```
sudo docker stack deploy -c docker-compose.app.yml app
```

Finally, you must configure kong, via API interface or using konga, to specify the following services:

```
{
    "protocol": "http",
    "name": "user",
    "retries": 5,
    "write_timeout": 60000,
    "id": "50164c90-1973-4e21-a4b5-19b3486d707d",
    "tls_verify": null,
    "tls_verify_depth": null,
    "enabled": true,
    "created_at": 1687376395,
    "path": null,
    "connect_timeout": 60000,
    "read_timeout": 60000,
    "client_certificate": null,
    "tags": null,
    "ca_certificates": null,
    "host": {YOUR_IP},
    "updated_at": 1713123030,
    "port": 8080
},
{
    "protocol": "http",
    "name": "container",
    "retries": 5,
    "write_timeout": 60000,
    "id": "ec4d990f-d93a-43fc-acc4-55a55dd203cf",
    "tls_verify": null,
    "tls_verify_depth": null,
    "enabled": true,
    "created_at": 1688404755,
    "path": null,
    "connect_timeout": 60000,
    "read_timeout": 60000,
    "client_certificate": null,
    "tags": [],
    "ca_certificates": null,
    "host": {YOUR_IP},
    "updated_at": 1707837117,
    "port": 8082
}
```

And using the available plugin define the keycloak configuration, such as:

```
{
    "id": "7602d9e7-3a07-4861-aa8a-f5a0bd1dc644",
    "consumer": null,
    "tags": null,
    "name": "oidc",
    "ordering": null,
    "created_at": 1688405411,
    "config": {
        "scope": "openid",
        "redirect_uri": null,
        "bearer_jwt_auth_signing_algs": [
            "RS256"
        ],
        "client_id": "kong",
        "bearer_jwt_auth_allowed_auds": null,
        "session_secret": null,
        "timeout": null,
        "skip_already_auth_requests": "no",
        "bearer_jwt_auth_enable": "yes",
        "groups_claim": "groups",
        "header_names": [],
        "header_claims": [],
        "disable_userinfo_header": "no",
        "userinfo_header_name": "X-USERINFO",
        "introspection_endpoint": "http://192.168.217.201:8081/auth/realms/haas/protocol/openid-connect/token/introspect",
        "realm": "kong",
        "validate_scope": "no",
        "disable_access_token_header": "no",
        "token_endpoint_auth_method": "client_secret_post",
        "access_token_header_name": "Authorization",
        "filters": null,
        "disable_id_token_header": "no",
```

```
        "bearer_only": "no",
        "id_token_header_name": "X-ID-Token",
        "ssl_verify": "no",
        "unauth_action": "deny",
        "recovery_page_path": null,
        "client_secret": "EBWvimLxbhYUy6u2p15lXg8hcrXO6ZEk",
        "introspection_endpoint_auth_method": null,
        "introspection_cache_ignore": "no",
        "response_type": "code",
        "use_jwks": "no",
        "ignore_auth_filters": null,
        "logout_path": "/logout",
        "revoke_tokens_on_logout": "no",
        "redirect_after_logout_uri": "/",
        "redirect_after_logout_with_id_token_hint": "no",
        "post_logout_redirect_uri": null,
        "discovery": "http://192.168.217.201:8081/auth/realms/haas/.well-known/openid-configuration",
        "access_token_as_bearer": "yes"
    },
    "instance_name": null,
    "protocols": [
        "grpc",
        "grpcs",
        "http",
        "https"
    ],
    "route": {
        "id": "c7ac038d-dfe8-4fe4-8893-a762c6615a00"
    },
    "enabled": true,
    "service": null
}
```


After that you will be able to use the entire application, via the REST API.

Utilization

Because although the application is not fully documented, the most available endpoints are included in the insomnia file, which can be imported into rest clients like insomnia or postman. The recommended use is through the developed client as it includes all the features developed to date, see https://gitlab.estig.ipb.pt/ipb1/estig/mi/thesis/2022_2023/hashcatasservice/client.

Feel free to explore the source code.

Authors

Carlos Lima

Jos&#xfffd; Rufino https://orcid.org/0000-0002-1344-8264

Rui Alves https://orcid.org/0000-0003-4128-8779

Apêndice C

Dockerfiles para criação de imagem

```
FROM golang:1.21.1-bullseye as builder
# Install app
WORKDIR /app

COPY  . .

RUN go mod download

RUN go build -o build/main src/cmd/main.go

#============================================#
# New Image
FROM ubuntu:22.04

# Install dependencies
RUN apt-get update && apt-get install -y --no-install-recommends gpg-agent \
    wget \
    clinfo \
    ocl-icd-opencl-dev \
    pkg-config \
    opencl-headers \
    make \
    build-essential \
    libssl-dev \
    libcurl4-openssl-dev \
    zlib1g-dev \
    pciutils \
    git \
```

```
    wget \
    ca-certificates \
    net-tools

ENV PATH /usr/local/nvidia/bin:${PATH}
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64:${LD_LIBRARY_PATH}
ENV OPENCL_VENDOR_PATH /etc/OpenCL/vendors

WORKDIR /tmp

RUN wget https://github.com/Kitware/CMake/releases/download/v3.20.0/cmake-3.20.0.tar.gz
    --no-check-certificate && \
        tar -zxvf cmake-3.20.0.tar.gz && \
        cd cmake-3.20.0 && \
        ./bootstrap && \
        make && make install

WORKDIR /root

RUN apt-get update && apt-get install -y --no-install-recommends mesa-common-dev

WORKDIR /opt

RUN git clone https://github.com/ruialves7/rOpenCL.git && \
        cd rOpenCL/ && \
        git checkout v1.1 && \
        cd code/rOpenCL_driver/build && \
        rm CMakeCache.txt && \
        cmake .. && \
        make && \
        mkdir -p /etc/OpenCL/vendors && \
        echo "/opt/rOpenCL/code/rOpenCL_driver/build/lib/CL/librOpenCL.so" >> /etc/OpenCL
    /vendors/ropencl.icd && \
        mkdir -p /etc/OpenCL/rOpenCL && cd /etc/OpenCL/rOpenCL

ENV HASHCAT_VERSION        v6.2.5
ENV HASHCAT_UTILS_VERSION  v1.9
ENV HCXTOOLS_VERSION       6.2.7
ENV HCXDUMPTOOL_VERSION    6.2.7
ENV HCXKEYS_VERSION        master

RUN update-pciids

WORKDIR /root
```

```
71
72 COPY hashcat ./hashcat
73 RUN cd hashcat && git checkout ${HASHCAT_VERSION} && make install -j4
74
75 RUN git clone https://github.com/hashcat/hashcat-utils.git && cd hashcat-utils/src && git
       checkout ${HASHCAT_UTILS_VERSION} && make
76 RUN ln -s /root/hashcat-utils/src/cap2hccapx.bin /usr/bin/cap2hccapx
77
78 RUN git clone https://github.com/ZerBea/hcxtools.git && cd hcxtools && git checkout ${
       HCXTOOLS_VERSION} && make install
79
80 RUN git clone https://github.com/ZerBea/hcxdumptool.git && cd hcxdumptool && git checkout
       ${HCXDUMPTOOL_VERSION} && make install
81
82 RUN git clone https://github.com/hashcat/kwprocessor.git && cd kwprocessor && git
       checkout ${HCXKEYS_VERSION} && make
83 RUN ln -s /root/kwprocessor/kwp /usr/bin/kwp
84
85 RUN mkdir -p /root/.local/share/hashcat && chmod -R 777 /root/.local/share/hashcat
86 RUN mkdir -p /root/aux && chmod -R 777 /root/aux
87
88 WORKDIR /root
89
90 RUN git clone https://github.com/philsmd/analyze_hc_restore.git
91
92 ENV ANALYZE_HC_FOLDER "/root/analyze_hc_restore"
93 ENV ROPENCL_HOSTS ""
94
95 WORKDIR /app
96 COPY --from=builder /app /app
97
98 EXPOSE 8080
99
100 RUN chmod +x /app/startup.sh
101
102 ENTRYPOINT ["/app/startup.sh"]
```

```
1 # New Image
2 FROM ubuntu:22.04
3
4 # Install dependencies
5 RUN apt-get update && apt-get install -y --no-install-recommends gpg-agent \
6     wget \
7     clinfo \
```

```
    ocl-icd-opencl-dev \
    pkg-config \
    opencl-headers \
    make \
    build-essential \
    libssl-dev \
    libcurl4-openssl-dev \
    zlib1g-dev \
    pciutils \
    git \
    wget \
    gawk \
    ca-certificates \
    net-tools

RUN rm -rf /var/lib/apt/lists/*

ENV PATH /usr/local/nvidia/bin:${PATH}
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64:${LD_LIBRARY_PATH}
ENV OPENCL_VENDOR_PATH /etc/OpenCL/vendors
ENV ROPENCL_HOSTS ""

WORKDIR /tmp

RUN wget https://github.com/Kitware/CMake/releases/download/v3.20.0/cmake-3.20.0.tar.gz
    --no-check-certificate && \
        tar -zxvf cmake-3.20.0.tar.gz && \
        cd cmake-3.20.0 && \
        ./bootstrap && \
        make && make install

WORKDIR /root

RUN apt-get update && apt-get install -y --no-install-recommends mesa-common-dev

WORKDIR /opt

RUN git clone https://github.com/ruialves7/rOpenCL.git && \
        cd rOpenCL/ && \
        git checkout v1.1 && \
        cd code/rOpenCL_driver/build && \
        rm CMakeCache.txt && \
        cmake .. && \
        make && \
```

```
        mkdir -p /etc/OpenCL/vendors && \
        echo "/opt/rOpenCL/code/rOpenCL_driver/build/lib/CL/librOpenCL.so" >> /etc/OpenCL/vendors/ropencl.icd && \
        mkdir -p /etc/OpenCL/rOpenCL && cd /etc/OpenCL/rOpenCL

ENV HASHCAT_VERSION        v6.2.5
ENV HASHCAT_UTILS_VERSION  v1.9
ENV HCXTOOLS_VERSION       6.2.7
ENV HCXDUMPTOOL_VERSION    6.2.7
ENV HCXKEYS_VERSION        master

RUN update-pciids

WORKDIR /root

COPY hashcat ./hashcat
RUN cd hashcat && git checkout ${HASHCAT_VERSION} && make install -j4

RUN git clone https://github.com/hashcat/hashcat-utils.git && cd hashcat-utils/src && git checkout ${HASHCAT_UTILS_VERSION} && make
RUN ln -s /root/hashcat-utils/src/cap2hccapx.bin /usr/bin/cap2hccapx

RUN git clone https://github.com/ZerBea/hcxtools.git && cd hcxtools && git checkout ${HCXTOOLS_VERSION} && make install

RUN git clone https://github.com/ZerBea/hcxdumptool.git && cd hcxdumptool && git checkout ${HCXDUMPTOOL_VERSION} && make install

RUN git clone https://github.com/hashcat/kwprocessor.git && cd kwprocessor && git checkout ${HCXKEYS_VERSION} && make
RUN ln -s /root/kwprocessor/kwp /usr/bin/kwp

RUN mkdir -p /root/.local/share/hashcat/sessions && chmod -R 777 /root/.local/share/hashcat
RUN mkdir -p /root/config && chmod -R 777 /root/config
RUN mkdir -p /root/files && chmod -R 777 /root/files

WORKDIR /root

COPY ./startup.sh /root/

RUN chmod +x /root/startup.sh

RUN git clone https://github.com/philsmd/analyze_hc_restore.git
```

```
FROM golang:1.21.1-bullseye

WORKDIR /app

COPY  . .

RUN go mod download

RUN go build -o build/users src/cmd/cmd.go

ENV MONGO_CONNECTION_STRING="mongodb+srv://api:3xcBW5Ngbg8h81fB@haas.r8s4vio.mongodb.net
    /?retryWrites=true&w=majority"
ENV KEYCLOAK_URI="http://192.168.217.201:8081/auth/"
ENV KEYCLOAK_REALM="haas"
ENV KEYCLOAK_USER="golang"
ENV KEYCLOAK_PASSWORD="golang"
ENV KEYCLOAK_DEFAULT_GROUP="2cd179f3-766e-4e46-be04-3345d186d7b3"
ENV KEYCLOAK_CLIENT_ID="auth_api"
ENV KEYCLOAK_CLIENT_SECRET="0srnNgOkFFVfALax7tkQ5uppiUxYrzZW"

CMD ["./build/users"]
```

Apêndice D

Arquivos de construção

```
version: '3.8'

services:
  users:
    image: carlossouzal/hashcloud-users:latest
    user: root
    deploy:
      replicas: 1
      restart_policy:
        condition: any
      placement:
        max_replicas_per_node: 1
    environment:
      MONGO_CONNECTION_STRING: "mongodb+srv://api@haas.r8s4vio.mongodb.net/?retryWrites=
    true&w=majority"
      KEYCLOAK_URI: http://192.168.217.201:8081/auth/
      KEYCLOAK_REALM: haas
      KEYCLOAK_USER: golang
      KEYCLOAK_PASSWORD: golang
      KEYCLOAK_DEFAULT_GROUP: 1ae93b35-e170-4dfe-8abc-6eabdcf57b5e
      KEYCLOAK_CLIENT_ID: auth_api
      KEYCLOAK_CLIENT_SECRET: t37VuFsrEGBZWS3PtiahdOjKIIwfqMjG
    ports:
      - '8080:8080'
    depends_on:
      - keycloak

  main:
```

```
    image: carlossouzal/hashcloud-main:latest
    user: root
    deploy:
      replicas: 1
      restart_policy:
        condition: any
      placement:
        max_replicas_per_node: 1
    environment:
      ROPENCL_HOSTS: "192.168.217.201\n192.168.217.208"
      MONGO_CONNECTION_STRING: "mongodb+srv://api@haas.r8s4vio.mongodb.net/?retryWrites=
    true&w=majority"
      KEYCLOAK_URI: http://192.168.217.201:8081/auth/
      KEYCLOAK_REALM: haas
      KEYCLOAK_USER: golang
      KEYCLOAK_PASSWORD: golang
      KEYCLOAK_DEFAULT_GROUP: 1ae93b35-e170-4dfe-8abc-6eabdcf57b5e
      KEYCLOAK_CLIENT_ID: auth_api
      KEYCLOAK_CLIENT_SECRET: t37VuFsrEGBZWS3PtiahdOjKIIwfqMjG
      REDIS_ADDR: "192.168.217.201:6379"
      REDIS_PASSWORD: ""
      REDIS_DB: 0
      BASE_FOLDER_VOLUME: "/mnt/hashcat"
      BASE_FOLDER_FILES_CONTAINER: "files"
      BASE_FOLDER_CONFIG_CONTAINER: "config"
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - /mnt/hashcat:/mnt/hashcat
    ports:
      - '8082:8082'
    depends_on:
      - keycloak
      - redis

  client:
    image: carlossouzal/hashcloud-client:latest
    user: root
    deploy:
      replicas: 1
      restart_policy:
        condition: any
      placement:
        max_replicas_per_node: 1
    ports:
```

```
      - '3000:3000'
    networks:
      - hashcloud-network
    depends_on:
      - keycloak
      - redis
      - users
      - main

networks:
  hashcloud-network:
    external: true
```

```
version: '3.8'

services:
  db:
    image: postgres:15.3-alpine
    restart: always
    environment:
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "postgres"
      POSTGRES_DB: "auth"
    ports:
      - '5432:5432'
    volumes:
      - db:/var/lib/postgresql/data
    networks:
      - postgres-network

  keycloak:
    image: jboss/keycloak:16.1.1
    restart: always
    environment:
      DB_VENDOR: postgres
      DB_ADDR: db
      DB_PORT: 5432
      DB_DATABASE: auth
      DB_SCHEMA: public
      DB_USER: postgres
      DB_PASSWORD: postgres
    ports:
      - "8081:8080"
    depends_on:
```

```
      - db
    networks:
      - postgres-network

networks:
  postgres-network:

volumes:
  db:
    driver: local
```

```
version: '3.8'

volumes:
  data: {}

networks:
  kong-net:
  hashcloud-network:
    external: true

services:
  kong-migrations:
    image: revomatico/docker-kong-oidc
    command: kong migrations bootstrap
    depends_on:
      - db
    environment:
      KONG_DATABASE: postgres
      KONG_PG_DATABASE: kong
      KONG_PG_HOST: db
      KONG_PG_USER: kong
      KONG_PG_PASSWORD: kong
      KONG_PLUGINS: bundled,oidc
    networks:
      - kong-net
      - hashcloud-network
    restart: on-failure
    deploy:
      restart_policy:
        condition: on-failure

  kong-migrations-up:
    image: revomatico/docker-kong-oidc
```

```
    command: kong migrations up && kong migrations finish
    depends_on:
      - db
      - kong-migrations
    environment:
      KONG_DATABASE: postgres
      KONG_PG_DATABASE: kong
      KONG_PG_HOST: db
      KONG_PG_USER: kong
      KONG_PG_PASSWORD: kong
      KONG_PLUGINS: bundled,oidc
    networks:
      - kong-net
      - hashcloud-network
    restart: on-failure
    deploy:
      restart_policy:
        condition: on-failure

  kong:
    image: revomatico/docker-kong-oidc
    user: "kong"
    depends_on:
      - db
      - kong-migrations-up
    environment:
      KONG_ADMIN_ACCESS_LOG: /dev/stdout
      KONG_ADMIN_ERROR_LOG: /dev/stderr
      KONG_ADMIN_LISTEN: '0.0.0.0:8001'
      KONG_CASSANDRA_CONTACT_POINTS: db
      KONG_DATABASE: postgres
      KONG_PG_DATABASE: kong
      KONG_PG_HOST: db
      KONG_PG_USER: kong
      KONG_PROXY_ACCESS_LOG: /dev/stdout
      KONG_PROXY_ERROR_LOG: /dev/stderr
      KONG_PG_PASSWORD: kong
      KONG_PLUGINS: bundled,oidc

    networks:
      - kong-net
      - hashcloud-network
    ports:
      - "8000:8000/tcp"
```

```
78      - "8001:8001/tcp"
79      - "8443:8443/tcp"
80      - "8444:8444/tcp"
81    healthcheck:
82      test: ["CMD", "kong", "health"]
83      interval: 10s
84      timeout: 10s
85      retries: 10
86    restart: on-failure
87    deploy:
88      restart_policy:
89        condition: on-failure
90
91  db:
92    image: postgres:9.6
93    environment:
94      POSTGRES_DB: kong
95      POSTGRES_USER: kong
96      POSTGRES_PASSWORD: kong
97    ports:
98      - '5555:5432'
99    healthcheck:
100     test: ["CMD", "pg_isready", "-U", "kong"]
101     interval: 30s
102     timeout: 30s
103     retries: 3
104   restart: on-failure
105   deploy:
106     restart_policy:
107       condition: on-failure
108   stdin_open: true
109   tty: true
110   networks:
111     - kong-net
112     - hashcloud-network
113   volumes:
114     - data:/var/lib/postgresql/data
115     - ./init-db-kong.sh:/docker-entrypoint-initdb.d/init-db.sh
116
117 konga-prepare:
118   image: pantsel/konga:latest
119   command: "-c prepare -a postgres -u postgresql://kong:kong@db:5432/konga_db"
120   networks:
121     - kong-net
```

```
      - hashcloud-network
    restart: on-failure
    depends_on:
      - db

  konga:
    image: pantsel/konga:latest
    restart: always
    networks:
      - kong-net
      - hashcloud-network
    environment:
      DB_ADAPTER: postgres
      DB_HOST: db
      DB_USER: kong
      DB_PASSWORD: kong
      TOKEN_SECRET: ahfdjgjgf79JKLFHJKh978953kgdfjkl
      DB_DATABASE: konga_db
    depends_on:
      - db
      - konga-prepare
    ports:
      - "1337:1337"
```

```
version: '3.8'

services:
  redis:
    image: 'redis:7.0.2'
    restart: always
    ports:
      - "6379:6379"
    volumes:
      - /mnt/hashcat/redis:/data
    command: redis-server --appendonly yes
    deploy:
      restart_policy:
        condition: on-failure
```

Apêndice E

Script de benchmarking

```python
import requests

def login(username, password):
    url="http://192.168.217.201:8000/users/signin"
    data = {
        "username": username,
        "password": password
    }
    headers = {"Content-Type": "application/json"}
    response = requests.post(url, json=data, headers=headers)
    return {
        "token": response.json()["access_token"],
        "type": response.json()["token_type"]
    }

def createHashcat(credential):
    url="http://192.168.217.201:8000/container/hashcat/"
    data = {
        "name": "Hashcat Name",
        "input": {
            "hash-type": 0,
            "attack-mode": 3,
            "mask": "",
            "input-file": {
                "content": "NDJmNzQ5YWRlN2Y5ZTE5NWJmN"
                +"Dc1ZjM3YTQ0Y2FmY2INCmRjNTk5YTk5NzJmZG"
                +"UzMDQ1ZGFiNTlkYmQxYWUxNzBi",
                "ext": ".hash"
```

```
            },
            "dict-file": [],
            "rule-file": None
        },
        "config": {
            "gpu_exclusive": "088a218b-b0c4-0993-4dcd-305b4710338b",
            "gpu_reserved": "",
            "use_shared": False
        }
    }
    headers = {
        "Content-Type": "application/json",
        "Authorization": credential["type"] + " "
            +  credential["token"],
    }
    response = requests.post(url, json=data, headers=headers)
    print(response.json())
    return



print("Starting...")
credentials = login("user", "user_password")
print("Signed...")
createHashcat(credentials)
print("Done...")
```