2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3315-2643-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/IPDPSW66978.2025.00137

HaaS - A Platform for Password Cracking in Distributed Heterogeneous Systems

Carlos Lima*, Rui Alves* and José Rufino†
*Instituto Politécnico de Bragança, 5300-253 Bragança, Portugal
Email: {carlos.lima,rui.alves}@ipb.pt
†Research Centre in Digitalization and Intelligent Robotics (CeDRI),
Laboratório Associado para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC),
Instituto Politécnico de Bragança, 5300-253 Bragança, Portugal
Email: rufino@ipb.pt ORCID: 0000-0002-1344-8264

***Abstract*—Traditional passwords and respective cryptographic hashes are still widely used for user authentication. Breaking these hashes, to recover the original passwords, may be necessary for a variety of legitimate reasons. Hashcat, a widely used password auditing tool, is able to exploit the parallel processing power of many GPUs to accelerate the breaking of cryptographic hashes. Moreover, there are already ways of performing this task in a distributed environment, though they face several challenges, including the difficulty of assembling and managing distributed deployments, and using them in a user-friendly and resource-efficient way. This work presents Hashcat-as-a-Service (HaaS), a novel platform that targets these challenges. It combines Hashcat with web technologies, containerization and remote OpenCL middleware, to allow the user-friendly management of Hashcat instances that leverage the processing power of distributed GPUs, including support of instances migration in order to maximize GPU utilization. The evaluation of HaaS in different configurations demonstrated promising results, confirming its ability to handle intensive workloads and its flexibility to adapt to different usage scenarios and resources availability, making HaaS a relevant contribution in the password recovery field.**

***Index Terms*—Web Development, Practical Cryptography, Parallel and Distributed Computing, Virtualization and Containerization, Heterogeneous Systems**

I. Introduction

Despite the growing importance of biometrics, hardware tokens, and multi-factor authentication (MFA), traditional passwords remain a fundamental component of user authentication [1]. They are primarily used due to their simplicity, familiarity, and ease of deployment. Unlike biometric authentication, which requires specialized hardware, or token-based authentication, dependent on physical devices, passwords are a universally accessible solution that requires no additional infrastructure beyond software implementation. Even in multi-factor authentication setups, passwords often serve as a critical first factor, reinforcing their ongoing relevance in digital security. This makes passwords particularly valuable for online services, enterprise systems, and applications that need a cost-effective and scalable method of securing user accounts.

This work was supported by national funds through FCT/MCTES (PIDDAC): UID/05757-Research Centre in Digitalization and Intelligent Robotics/ Centro de Investigação em Digitalização e Robótica Inteligente (CeDRI).

Authentication passwords are not supposed to be stored in clear-text in the target systems. Instead, cryptographic hash functions transform them in one-way irreversible hashes, for storage purposes and authentication validation. Due to their importance in the authentication process, it is fundamental to use strong passwords and robust cryptographic hash functions. There are, however, legitimate reasons to try to break a cryptographic hash, recovering the original password.

Advanced hash analysis tools, specially when integrated with artificial intelligence techniques [2] or parallel/distributed computing platforms [3], offer powerful solutions to assert the passwords robustness. Hashcat [4], widely used for breaking passwords recovery, is a good example of such tools, once it can accelerate the breaking of cryptographic hashes using the computing power provided by heterogeneous systems.

In particular, Hashcat is able, by design, to take advantage of many GPUs simultaneously. Hashcat's performance, though, ends up being limited by the amount and type of GPUs available in the host system. However, computing clusters and cloud computing platforms are nowadays populated with GPUs (a trend that accelerated after the emergence of AI training and AI-based applications), meaning that tools already prepared to use more than one GPU would greatly benefit from mechanisms capable of exposing them to distributed devices.

It is also possible that the Hashcat workloads are unable to saturate the GPUs, which is mostly certain when GPUs are remote and commodity network interconnects are used, instead of advanced communication fabrics and protocols (like Infiniband and RDMA [5]). This opens up other opportunities, namely the possibility of migrating workloads to become near the most performant GPUs, or sharing GPUs by different and independent Hashcat workloads. Such orchestration should allow for a much more efficient use of the visible GPU set.

This paper presents Hashcat as a Service (HaaS), a platform that aims to simplify and optimize password cracking with many distributed GPUs, making it user-friendly and to efficiently use the GPU set. The HaaS platform combines Hashcat with web technologies, containerization and remote OpenCL (rOpenCL) middleware [6] to achieve these goals, providing an alternative to other forms of distributed execution of Hashcat.

2995-066X/25/$31.00 ©2025 IEEE
DOI 10.1109/IPDPSW66978.2025.00137

The remainder of the paper is organized as follows: section II contextualizes this research and references related work; section III describes the HaaS architecture and implementation; section IV provides the results of a preliminary evaluation of HaaS; lastly, section V concludes and defines future work.

II. Background and Related Work

Passwords are still the main user authentication mechanisms. Robust cryptographic hash functions allow the generation of unique non-reversible hashes from clear-text passwords. Hashes associated with user accounts are stored in protected system files or databases that are queried during authentication procedures. Even though hashes are supposed to be mathematically irreversible, there are several techniques and tools that can be used to recover the original passwords, and many legitimate reasons to do so (cybercrime investigation, security audits, data recovery, academic research, etc.).

Some advances in cracking passwords come from studies carried out some time ago, such as using standard Markov modeling techniques from natural language processing, to dramatically reduce the size of the password space to be searched [7]. Another technique, proposed by Weir et al [8], involves the automatic creation of a probabilistic context-free grammar based on a training set of known passwords, which allows to generate word-mangling rules, and from these, password guesses to be used by password cracking tools.

AI techniques have also been used to break passwords, with the PassGAN tool being one of the main contributions, based on the use of a Generative Adversarial Network (GAN) to autonomously learn the distribution of real passwords from actual password leaks, and to generate high-quality password guesses [2]. More recently, in a completely different direction, Zhou et al introduced password cracking using chunk similarity [9], whereby passwords are seen as a sequence of chunks (sub-strings) that appears frequently in a password dataset, and new passwords are generated by choosing a chunk and replacing it with another one according to their similarities.

Breaking algorithms and techniques find expression in many tools, available to both malicious and legitimate actors. THC Hydra [10], Aircrack-ng [11], NCrack [12], John the Ripper [13], and Hashcat [4] are some of the most used and well-known tools to break cryptographic hashes. In particular, Hashcat, used in this work, claims to be the fastest offline tool of its kind, due its capability of exploiting multiple GPUs.

GPUs are particularly efficient at tasks that benefit from massive parallel processing power, such as password-cracking attacks, where multiple attempts are performed simultaneously. Tools like Hashcat take advantage of this feature of GPUs, to considerably accelerate the cracking process. Moreover, Hashcat is also able to use multi-core CPUs, and supports several compute runtimes (e.g., NVidia OpenCL and CUDA, Intel OpenCL, etc.), highlighting the importance of heterogeneous systems in the cyber security domain and password cracking.

In order to further improve the performance of these tools, several solutions have demonstrated the benefits of their use in a distributed configuration. This makes possible to test a larger number of passwords or hash combinations in parallel. An early example [3] of this approach combined the use of CUDA and MPI. Another, exploited the execution of the Fitcrack application for password recovery in nodes connected through BOINC, a software platform for computing using volunteered resources (personal computers) around the world [14].

Recently, cloud instances with GPUs have also been used to perform distributed executions of Hashcat through Hashtopolis [15], [16], which encapsulates Hashcat and offers additional facilities, like a web based interface and multi-user support, also present in the HaaS platform introduced in this paper.

HaaS, however, does not explicitly divide the work by different tasks, dispatched to different distributed agents with GPUs attached, like Hashtopolis. Instead, a HaaS workload is a containerized Hashcat instance, that attacks one or more cryptographic hashes, with access to a GPU set that may be changed in runtime, and include local and/or remote GPUs (with the latter being accessed transparently); also, GPUs may be shared by several workloads (maximizing their usage) or used by one workload at a time (minimizing turnaround time); moreover, workloads may be migrated, in order to move them closer to specific GPUs; combined, these features provide a high degree of flexibility and efficiency in the use of the GPUs.

The HaaS platform was thus developed with the aim of optimizing the execution of Hashcat, maximizing the use of available computing resources, regardless of their location. One of the central aspects of this flexibility is the use of rOpenCL [6], a middleware layer that allows pre-compiled OpenCL applications to transparently access remote GPUs. In the context of HaaS, though, due to the containerized nature of Hashcat workloads, and the possibility of migration to other hosts, local GPUs are also accessed through rOpenCL, meaning that the view of the GPU set is always consistent, even after the migration of a workload (single-system-image).

The use of rOpenCL also ensures network-portability, once its communication layer is built on BSD sockets and only requires a TCP/IP network stack, which is today ubiquitous. Therefore, no expensive/proprietary network fabrics are necessary to allow an efficient acess of remote GPUs in HaaS. The direct integration of rOpenCL with the OpenCL ICD (Installable Client Driver) Loader ensures that interactions with remote devices are transparent and compatible with existing pre-compiled OpenCL applications, like Hashcat.

Indeed, Hashcat itself has already been combined with rOpenCL [6], demonstrating its ability to reduce Hashcat execution time through the contribution of remote GPUs. In addition, rOpenCL was also previously integrated in container orchestration environments [17], where it showed its ability to significantly reduce the execution times of workloads sharing both local and remote GPUs. This made rOpenCL a natural choice as a base component of HaaS, though other well-known alternatives, like PoCL-R [18], could also have been adopted.

III. HaaS Architecture and Implementation

The HaaS platform combines Hashcat with container technologies (via Docker Swarm) and the rOpenCL middleware to

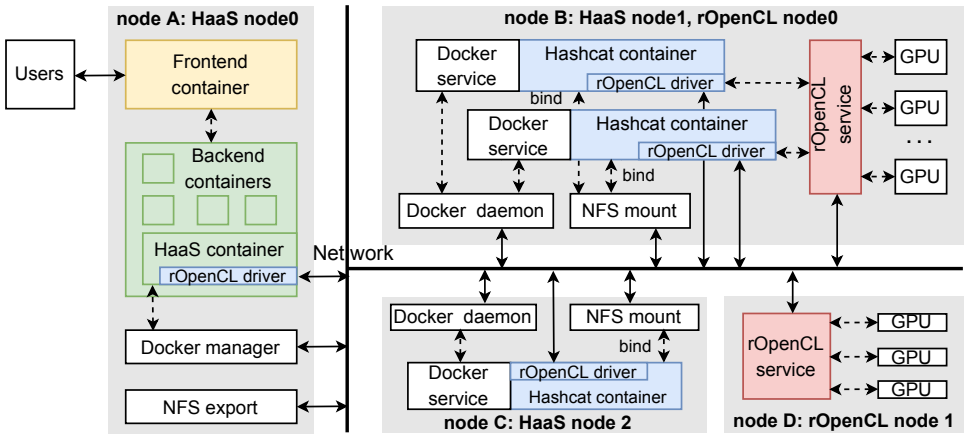


Fig. 1. A typical deployment of the HaaS architecture

create a web-exposed software service (Hashcat-as-a-Service) that is user-friendly, and flexible and efficient in the way the underlying hardware resources are used. Figure 1 shows a typical HaaS deployment, where it is possible to identify important architectural components and interactions, next discussed.

To start with, there are three types of nodes (hosts) involved, some of which may overlap: a single *HaaS frontend node* (node A), one or more *HaaS backend nodes* (nodes B and C), and one or more *rOpenCL backend nodes* (nodes B and D). HaaS backend nodes are able to run Hashcat containers, while rOpenCL backend nodes have local shareable GPUs.

Considering these types of nodes one can define the concepts of *HaaS cluster* and *rOpenCL cluster*; here, nodes A to C form an HaaS cluster, and nodes B and D form an rOpenCL cluster, with both clusters overlapping in node B. Moreover, because Docker Swarm is used to manage the various containers created in the nodes of the HaaS cluster, the HaaS cluster and the Docker Swarm cluster fully overlap.

Users interact with HaaS through a web-based interface, provided by a frontend container hosted in the HaaS frontend node. In this container, the web interface is implemented using a combination of Node.js, React, and Next.js technologies.

The user requests are processed and supported by a set of backend containers based on modern and well proven technologies, including the Kong API Gateway (integrated with a PostgreSQL database), Keycloak (for identity and access management, also integrated with the same PostgreSQL database), MongoDB (used to store metadata on Hashcat instances), and Redis (used to manage GPU inventory and reservations); the mediation between users and the Hashcat containers is performed by a special backend container – the HaaS container – through the local Docker Swarm manager service.

To support migration of Hashcat containers (which requires access to their checkpointed state in the target node), a NFS share is created in the HaaS frontend node, and exported to all HaaS backend nodes, where it is mounted and exposed (via the bind mount mechanism) to the local Hashcat containers.

Each HaaS backend node hosts, at any time, a set of Hashcat containers. Each one of these containers was created to run a single Hashcat instance (workload) on the behalf of a user, using a certain subset of the global set of GPUs exposed by the rOpenCL driver. Some of the GPUs used by a Hashcat container may be local, meaning there is an instance of the rOpenCL service running in the same host of the container, managing the access to the local GPUs; some of the GPUs used may be remote, with the rOpenCL driver being able to dispatch OpenCL requests, from Hashcat instances, to the remote rOpenCL service that manages the remote GPUs required. An HaaS backend node do not necessary have local GPUs (as in node C), meaning all Hashcat containers locally created will be limited to the use of remote GPUs.

The initial creation of a Hashcat container in a specific HaaS backend node is decided by the Docker Swarm load balancing mechanism, but under certain restrictions related with specific properties of the GPUs required by the user (local vs remote locality, and shared vs exclusive used), to be discussed bellow, along with a clarification of other important technical details.

As previously defined, a *workload* is an instance of the Hashcat application running inside a *container* created for that purpose. However, for simplicity, both concepts (workload vs Hashcat container) will be used from now interchangeably.

A. GPUs Management

When the HaaS container starts in the HaaS frontend node, the global list of GPUs exposed by rOpenCL is learnt using the `clinfo --json` command. These GPUs are registered in Redis, and the HaaS platform administrator explicitly defines their usage mode, which can either be *exclusive* or *shared*.

Exclusive GPUs can only be used by a single workload at a time. Therefore, they can only be assigned to a workload if they are not in use. Additionally, each workload can use at most one exclusive GPU (to prevent a single user from consuming all available GPUs, which could starve other services and workloads.) and this GPU must be local (attached to the same host where the workload is running). A workload

may choose to use only one exclusive GPU or combine it with shared GPUs. In the current HaaS implementation, shared GPUs are not individually manageable by users or individually assignable to workloads; instead, they are all shared among all workloads that opted to use that kind of GPUs. However, they can be local or remote (though always exposed by rOpenCL).

Each user is responsible for selecting the exclusive GPUs for its own workloads, as well as deciding if they will use shared GPUs. This selection can be dynamically modified throughout the workloads lifetime. At any time, a workload without shared GPUs can have them assigned. Additionally, shared GPUs can be detached from a workload, provided it has at least an exclusive GPU assigned. A workload can also be assigned an exclusive GPU later during its execution (if doesn't have already one assigned), or be detached from it (provided it has the shared GPUs assigned). If an exclusive GPU is currently busy, requests for it will be put in a FIFO queue specific to the GPU (see *GPUs Reservation* for details).

Changing the GPU set of a workload is possible due to the native checkpointing mechanism of the Hashcat application; this mechanism supports assynchronous checkpoint requests that are checked and honored by Hashcat at specific execution points, generating a state file that can be used to later restart and continue the Hashcat application execution. By carefully tweaking the state file, the GPU set can be changed transparently, so when Hashcat restarts it begins using the new set.

GPUs Reservation: When there are no exclusive GPUs available (i.e., there are GPUs tagged as exclusive, but all are busy), a user can still request such a GPU for a workload. The request will enter a FIFO queue specific to that exclusive GPU, that holds all requests for it. When the GPU becomes free, the workload at the head of the queue is assigned the GPU, and is removed from the queue; if the workload is already in progress (using shared GPUs), it will be paused, its GPU set changed, and restarted (using the Hashcat checkpointing mechanism); if the workload never started (once it asked for this specific exclusive GPU), it will now be allowed to begin its execution.

It may also happen an enqueued workload finishes before the exclusive GPU becomes free (once it has been using the shared GPUs), in which case the workload is just dequeued.

B. Workloads Migration

The Hashcat workloads may be migrated between nodes of the HaaS cluster (which, as already stated, are the same nodes that form the Docker Swarm cluster). This only takes place when there is a change in the GPU set of the workload, namely when the user adds an exclusive GPU, or decides to switch an exclusive GPU for another that is placed in a different node.

This relationship between the migration mechanism and exclusive GPUs stems from an important architectural restriction of HaaS: exclusive GPUs can only be local, as remote, exclusive GPUs would be severely underutilized.

The migration process takes advantage of the Docker Swarm scheduler, which already has attributes capable of suggesting the best allocation of a service, since each node has defined the available local GPUs in its labels.

Migrating a workload ends up being a very efficient and straightforward process, once there is no migration of the full hosting container (a Hashcat container); instead, once the workload has been checkpointed, its state is stored in the NFS share used throughout the HaaS deployment; the old Hashcat container is destroyed and a new one is created in another HaaS backend node, with the help of the Docker Swarm facilities; then, the state of the old container is pulled from the NFS share, tweaked to reflect the new exclusive GPU, and the workload is restarted; all shared GPUs that the workload was already using will still be accessible, some of them becoming remote, and maybe others becoming local, but all of them still being accessed transparently through the rOpenCL layer.

C. User Interface and Actions

The HaaS web interface is the platform entry point. Beyond the usual operations related to user-management (creation of accounts, passwords change and login/logout), it supports several operations on GPUs and workloads, described next.

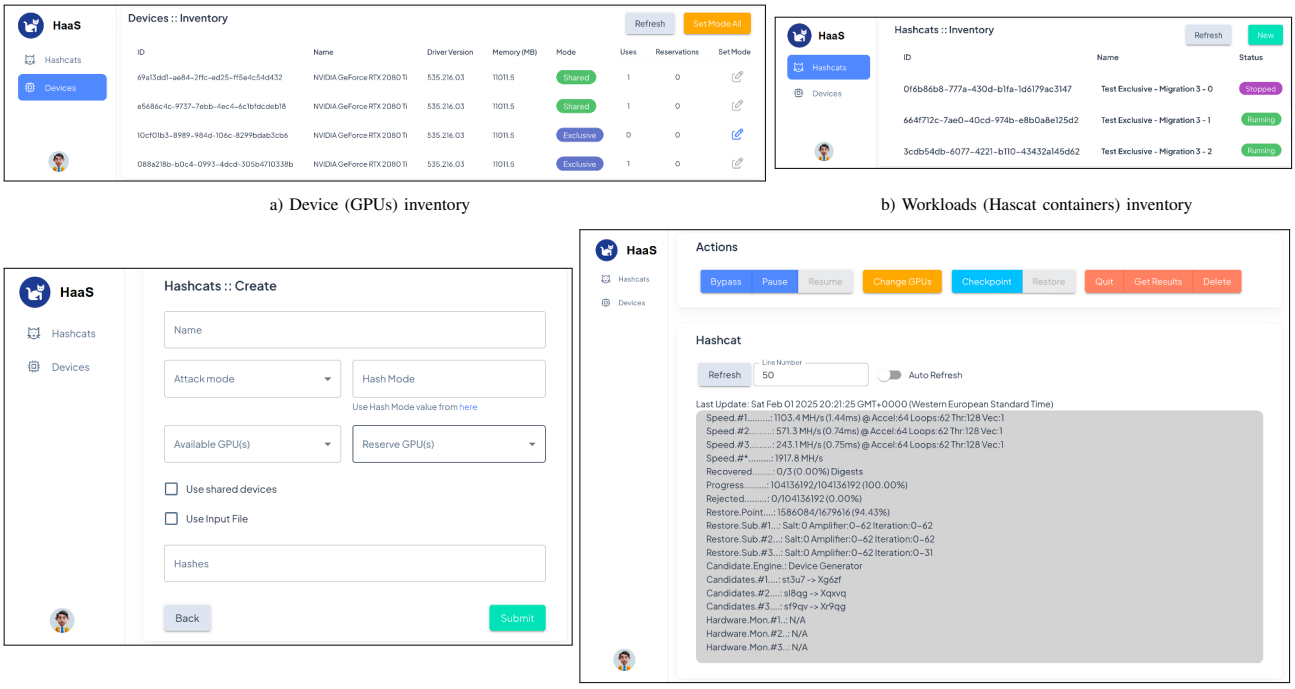
A GPU inventory panel (see Figure 2.a) shows, for each GPU, its platform internal identifier (ID), user-friendly Name (brand and model), RAM amount, usage Mode (exclusive vs shared), number of workloads using it (Uses), and number of Reservations (exclusive GPUs only); administrative users can change (SetMode) the usage mode of free GPUs.

A workloads inventory panel (see Figure 2.b) shows all (administrative view) or user-specific workloads; for each workload it displays its platform internal identifier (ID), user-friendly Name, and current Status (Running/Stopped); clicking on a workload opens up a workload-specific panel; new workloads may also be created through the New button.

When creating a new workload (see Figure 2.c), the user has to specify the workload user-friendly Name, Attack Mode and Hash Mode (using Hashcat specific conventions), which exclusive GPUs to use immediately from those available (and which ones to reserve from those currently unavailable), and if shared GPUs should be used or not; additionally, the hashes to crack are supplied through a file or explicitly enumerated.

After a workload has been created, a workload-specific panel allows several actions on it, depending on its current state (Running or Stopped). All operations supported by Hashcat in a command-line environment, through specific keystrokes, are also exposed and actionable in the GUI of the workload. For instance, for a running workload (see Figure 2.d), it is possible to pause the current attack or just bypass it (skip it and move to the next attack), change the GPU set used (possibly triggering a migration of the workload), force a state checkpoint, get (download) the current results (recovered passwords so far), quit (terminate) the Hashcat execution, or remove it (terminate and delete the enclosing container).

The user may also follow, in real time, the output produced by an Hashcat instance. In the Hashcat container, the Hashcat application redirects its output to a specific file in the HaaS NFS share, and the backend HaaS container makes the updated lines from that file available to the frontend container.



a) Device (GPUs) inventory

b) Workloads (Hascat containers) inventory

c) Workload creation

d) Workload running

Fig. 2. Some screens of the HaaS GUI

IV. Experimental Evaluation

This section presents the tests carried out to evaluate the behavior of HaaS in various usage scenarios. As HaaS is a platform designed to run GPU-intensive workloads, it is essential to understand how different GPU configurations and usage modes impact the platform's performance and stability.

A. Evaluation Environment

The testbed consisted of three Linux Ubuntu 22.04 64 bits virtual machines (VM0, VM1, and VM2), hosted on separate identical servers of a KVM-based virtualization cluster. VM0 was always used as the HaaS frontend node (node A in Figure 1), while VM1 and VM2 were used as HaaS backend nodes. In all tests, VM1 adopted the profile of node B in Figure 1, while VM2 adopted the same profile in some tests, and the profile of node D in others (this will be clarified as needed).

VMs 1 and 2 were assigned 32 vCores (from an AMD EPYC 7452 CPU), 64 GB of vRAM (from DDR4 2666 MHz ECC RAM), 512 GB of vDisk (from a NVMe PCIe 3.1 U.2 SSD), a 100Gbps vNIC (Linux bridge, on top of a Mellanox ConnectX-5 EN adapter), and 2 NVIDIA RTX 2080 Ti 11GB GPUs via *passthrough* (these are not recent GPUs, but it was the largest homogeneous set available in the KVM cluster at the time). VM 0 had similar virtual hardware, though scaled down and without GPUs. All VMs had the same version of the auxiliary tools and drivers they used in common: Hashcat 6.2.5, Docker 25.0.3, rOpenCL 1.1, and NVIDIA Drivers 535.

B. Methodology

The evaluation was based on the execution, under different conditions, of up to 3 identical workloads (this is the maximum number of identical workloads that could fit in the global memory of the testbed GPUs, when used in shared mode).

Each workload breaks the same set of 3 hashes (see bellow). Although different workloads can be expected in a real-world scenario, choosing identical workloads for benchmarking, and using only one type of GPUs, provides a uniform evaluation environment, which makes it easier to draw conclusions.

Hashcat was executed in brute force mode, using 20800 as the hash parameter [4], with the passwords being first submitted to the MD5 algorithm and then to SHA256 [19]. The 3 passwords (and respective hashes) were: Test1234 (2bb65a50112c1d8c2836c3cda4a1a6338dfce0c17c1f5c9fa57181d504a15e5), Password (e765e47fd32132c38dd4826f75556fc09038d775bf705f76c147b352362695b), and P455w0rd (65e9ae1761e1473afe9cdcac1ea9ceeca2f1346093621c6fac20f032c159813d).

To provide a comprehensive assessment of HaaS's performance, several metrics were evaluated: lifetime of 1 workload, startup latency with 3 workloads, and execution time of 3 workloads (with and without workload migrations involved).

The scenarios tested involve different GPU configurations – specific combinations of the number of GPUs used (1 to 4), their location (local and/or remote), and their usage modes (exclusive vs shared). Each GPU configuration is tested 4 times and, depending on the metric evaluated, its values are considered individually or averaged (arithmetic average). Only

the 3 best values (out of the 4 collected) are averaged: the 4th worst value was observed to be either similar to the other 3, or an outlier, in the latter case most probably due to the testbed being deployed in a shared virtualization environment.

C. Lifetime of a Hashcat Container

In the first test, only VM1 is used as a HaaS backend node. The test measured the lifetime of Hashcat containers in a basic scenario: a single Hashcat container executes in VM1, using a single GPU (local to VM1, in exclusive or shared mode).

Here, the lifetime of a Hashcat container is the time elapsed from the request to create it until it is destroyed. This lifetime thus includes: a) *startup latency* – time from the request to create the container to the startup of the Hashcat application on it; b) *execution time* – time from the startup of the Hashcat application on the container, to the termination of that application (time spent breaking the 3 hashes); c) *destruction latency* – time from the termination of the Hashcat application on the container, to the destruction (removal) of that container.

The results (averages of the best 3 runs) of this test were: lifetime in shared mode - 63s; lifetime in exclusive mode - 63.7s. These values are similar, though slightly better in shared mode, because in exclusive mode there is additional code that has to be executed to ensure the exclusive use of the GPU. But, apart from that, as in this case there is no competition for the use of the GPU by more than one container, different GPU execution modes have almost no effect on performance.

This allows to anticipate that any noticeable differences in performance shown by the two modes in more complex scenarios, will have causes other than those internal to the modes (i.e., one mode having been implemented more efficiently than the other, which would introduce a bias in the results).

D. Startup Latency with many Hashcat Containers

The second set of tests assessed the startup latency when several Hashcat containers are requested to be created, still limited to the use of a single GPU, now usable in the following configurations: if in exclusive mode, the GPU will be local; if in shared mode, the GPU can be local or remote; if local, besides VM0, only VM1 will be involved (under profile B); if the GPU is remote, VM2 will also be used (in profile D).

For each GPU usage mode (shared local, shared remote, and exclusive local), these tests were done in 4 consecutive rounds. In each round, 3 workloads (W1, W2, W3) were launched and the startup latency of each workload was recorded. The results are shown in Figure 3. Given the orders of magnitude of the latencies involved, the latencies for the exclusive local mode are represented against the right vertical axis. Points 1 to 3 on the horizontal axis represent the 3 workloads of the 1st round, points 4 to 6 represent those of the 2nd round, and so on.

There is a significant disparity between the startup latencies with the exclusive mode and with the shared modes. This difference stems from the fact that, while in shared mode the Hashcat containers are created immediately after the request is received, in exclusive mode each request has to wait for

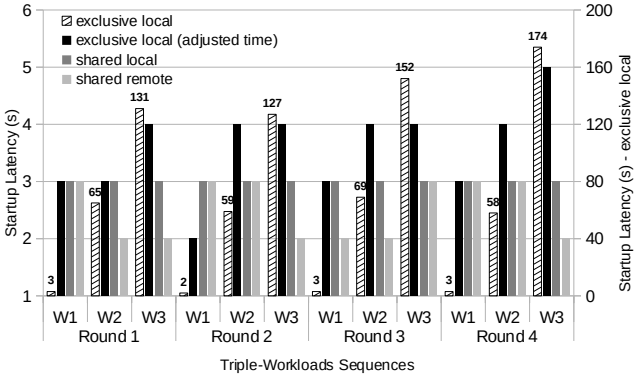


Fig. 3. Startup Latency of 3 workloads with 1 GPU in different usage modes.

the GPU to become available, which only happens after the previous workload that was using the GPU has finished.

It is also noticeable that the startup latency of the shared mode with a remote GPU is sometimes slightly slower than with a local GPU. This is purely accidental for the specific tests perfomed, once the reverse behavior can be observed when repeating the tests a few more times. The reason is that in both cases a local Hashcat container is being created, and it is only the latency of that creation that is being measured; it is only when the Hashcat application starts running that there will be a significant impact of the locality of the GPUs.

In order to understand how much time is actually spent launching and starting a Hashcat container in exclusive mode, it is necessary to subtract the *waiting time* (in the GPU queue) from the *startup latency*, resulting in an *adjusted startup latency*. Thus, Figure 3 also shows this adjusted latency for exclusive mode (black bars), which ends up being close to the latencies of the shared modes, although slightly bigger due to the need to manage the queue associated with the GPU.

Based on this analysis, it can be concluded that the time taken to create Hashcat containers is relatively independent of their configuration. With regard to the time taken to destroy them (*destruction latency*), although no values are presented, they are also relatively consistent for the various execution modes and GPU locations. As a result, these two latencies are not relevant for evaluating the performance of the various configurations, and will therefore be ignored in the next benchmarks presented, which focus solely on *execution time*.

E. Execution Time without Migration

In the third set of tests, the *execution time* of the 3 workloads was measured for all GPU configurations allowed when VM1 is in profile B and VM2 is in profile D, meaning workloads must always be created in VM1 and cannot be migrated, and they can use local (from VM1) and/or remote (from VM2) GPUs. Table I shows the GPU configurations tested, which also take into account the restrictions of the current HaaS implementation and the composition of the GPU set of testbed. Each workload will want to use the same GPUs of

TABLE I
GPU CONFIGURATIONS TESTED WITHOUT MIGRATION.

Locality			*Remote*		
	mode		**NA**	**shared**	
		GPUs	*0*	*1*	*2*
Local	**NA**	*0*	-	1Rs	2Rs
	shared	*1*	1Ls	1Ls+1Rs	1Ls+2Rs
		2	2Ls	2Ls+1Rs	2Ls+2Rs
	exclusive	*1*	1Le	1Le+1Rs	1Le+2Rs
		2	2Le	2Le+1Rs	2Le+2Rs
	shared and exclusive	*2*	1Ls1Le	1Ls1Le+1Rs	1Ls1Le+2Rs

each configuration, which will be possible simultaneously for shared GPUs, but will have to be done in turns for exclusives.

In the table, the GPU configurations are described by a notation that identifies the number of GPUs involved, their Locality from the workloads point of view (L=Local, R=Remote) and their use mode (s=shared, e=exclusive); the symbol "+" separates Local from Remote GPUs in the same configuration.

Again, it is recalled the restriction of the current HaaS implementation, by which no more than one exclusive GPU can be used per each Hashcat container (see section III-A). Thus, it makes no sense to test configurations with more than three exclusive GPUs, as any additional GPU in this mode would not be used, given that the maximum number of Hashcat containers running simultaneously in our testbed is also 3.

It is also recalled that if a Hashcat container uses an exclusive GPU, that GPU must be local. This means a maximum of 2 containers that use exclusive GPUs can co-exist on each backend node of the testbed, since there are 2 GPUs per node. In addition, as there are 2 nodes with 2 GPUs each, an overall maximum of 4 containers may exist, each with its own exclusive GPU. However, as explained before, the maximum number of simultaneous containers the testbed supports is 3.

The results of these tests (averages of the best 3 runs with 3 workloads) are next discussed, for different GPU localities.

1) Local GPUs only: Figure 4 shows the *execution times* of the 3 workloads when they use only local GPUs, for all allowed combinations of use modes (the same figure also shows results for remote-only GPUs, to be discussed later).

There are two types of time measured: a *global time* and an *individual time*. The global time is the time that elapses between the moment Hashcat starts executing in the 1st container to start it, and the moment Hashcat finishes in the last container to finish it (note that these containers can be different); it is therefore the execution time of all 3 containers. The individual time corresponds to the average execution time from the perspective of each container: the time that elapses from the moment when the container starts executing Hashcat until it finishes; this time is always less than the global time (but close when the GPUs are used in shared mode).

Comparing the global times of the local shared (1Ls, 2Ls) and local exclusive (1Le, 2Le) scenarios, one concludes:

- shared GPUs allows faster execution of the 3 workloads set than exclusive GPUs: speedup of 186/148 = 1.26 with one GPU, and of 123/90 = 1.37 with two GPUs;

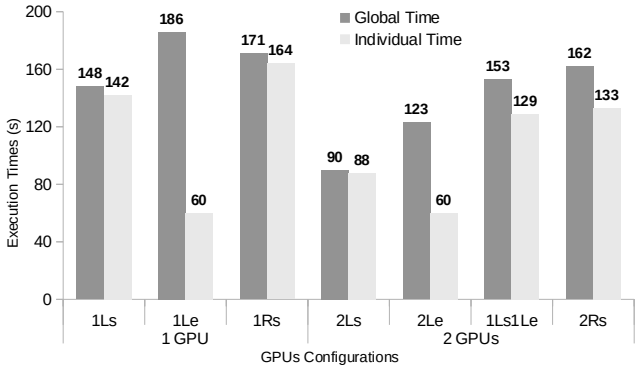


Fig. 4. Execution times (global and individual average) of 3 homogeneous workloads without migration, and with local-only or remote-only GPUs.

- in both modes, adding one more local GPU decreases (as expected) the global execution time; true, absolute times are different between modes, but the speedups of an extra GPU are close: speedup of 148/90 = 1.64 with two shared GPUs, and 186 / 123 = 1.51 with two exclusive GPUs;
- mixing local GPUs in different modes (1Ls1Le) does not yield any benefit over using only shared GPUs (even if only one GPU - 1Ls), meaning there seems to be some kind of advantage of shared GPUs over exclusive ones.

The fact that shared GPUs (1Ls, 2Ls) ensure faster execution than exclusives (1Le, 2Le) may seem counter-intuitive: one of the premises for the creation of the exclusive mode was that it should favor performance. And, looking at the individual execution times, this is the case: from its own perspective, each container executes faster with an exclusive GPU than with a shared one. So, what is the reason for the execution of the workload set to be globally faster in shared mode? The answer lies in the analysis of the GPU(s) utilization.

Figure 5 shows the load (%) of one GPU (measured by the nvidia-smi utility) during the 1st minute in which it's executing the 3 workloads in shared mode, and during the 1st minute of execution of the same workloads in exclusive mode (the 1st workload finishes after one minute, as already shown in Figure 4). It is clear that in exclusive mode the GPU is used less efficiently, having room for extra load that is not being demanded. Thus, by sharing the GPU among several workloads, the workload set finishes faster, compared to the sequential execution of the workloads in exclusive mode.

For individual times, Figure 4 shows that adding one more local exclusive GPU (1Le→2Le) doesn't change them (60s): the GPUs are equal, so its irrelevant on which a workload runs. Adding one more local shared GPU (1Ls→2Ls) helps reducing not only the global, but also the individual time (speedup=142/88=1 .61, similar to the one of the global times): with more GPUs, there are more opportunities for a workload to progress when the others aren't using the same GPUs.

2) Remote GPUs only: Another use case allowed by HaaS is to exploit only remote GPUs, which currently is supported only if they are all used in shared mode. Figure 4 shows also the corresponding execution times (1Rs and 2Rs bars).

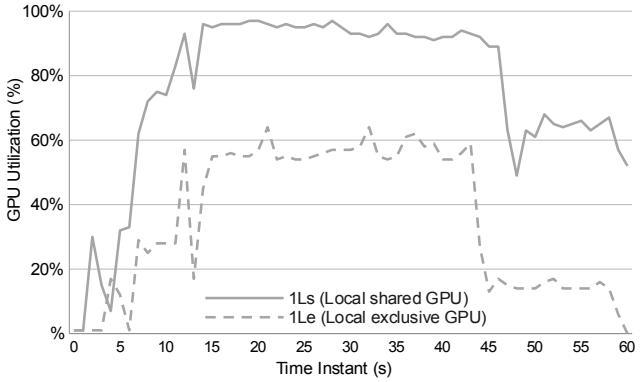


Fig. 5. GPU load with 3 workloads in the 1Ls and 1Le scenarios (1st minute).

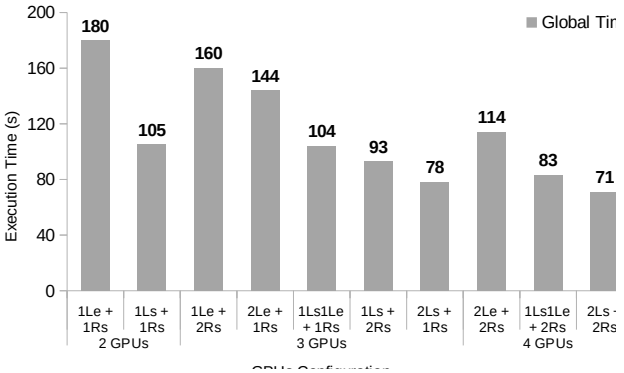


Fig. 6. Execution times (global) without migration, for hybrid scenarios.

Using shared remote GPUs shows a similar trend to that already observed with shared local GPUs: adding a 2nd GPU boosts performance. However, with remote GPUs, the global speedup generated by this addition is only 171/162=1.06, much lower than with local GPUs (148/90=1.64). Furthermore, it is expected that using only remote GPUs offers lower performance than an equal number of local GPUs: in this specific testbed, using 1 local GPU for the 3 workloads is 171/148=1.15 times faster that using 1 remote GPU for the same purpose, and using 2 local GPUs is 162/90=1.8x faster.

The individual times of the workloads with remote shared GPUs also follow the same trend of the individual times with local shared GPUs: they are lower than, but (relatively) close to the global times. Of course, due to the remote nature of the GPUs, the remote execution times are comparatively higher.

The explanation for a lesser performance when using only remote GPUs, versus using only local GPUs, lies in the penalty imposed by the network exchanges conducted by the rOpenCL middleware in order to redirect all OpenCL requests generated by the workloads running on VM1, to VM2 – which hosts the remote GPUs. Also, the fact that adding a 2nd remote GPU produces an anemic speedup (1.12) is also related to the increasing of the rOpenCL communication load, limiting the acceleration provided by the addition of extra remote GPUs.

3) Local and Remote GPUs: Still without involving any workloads migration, there are also hybrid scenarios where local and remote GPUs are both present in the GPU configuration of the scenario. The hybrid scenarios supported by our testbed are those whose name includes the sign "+" in Table I, involving 2, 3 or 4 GPUs; here, remote GPUs can only be shared, and local GPUs can be shared, exclusive, or both.

The global execution times for the hybrid scenarios are shown in Figure 6 (the individual times will be ignored henceforth). By observing it, some conclusions can be drawn:

- several hybrid scenarios improve the performance over the best non-hybrid (which is 2Ls, with 90s of global time – recall Figure 4): the 3 GPUs configuration 2Ls+1Rs (78s), and the 4 GPUs configurations 1Ls1Le+2Rs (83s) and 2Ls+2Rs (71s); the 3 GPUs configuration 1Ls+2Rs (93s) offers comparable performance to 2Ls (93s $\approx$ 90s);
- the best hybrid scenario (2Ls+2Rs, 71s) yields a speedup of 90/71=1,27 over the best non-hybrid scenario (2Ls, 90s); this is a small gain, considering that the number of GPUs involved doubles (from 2 to 4); however, the extra GPUs are remote, which explains the weak scalability;
- for each set of configurations in which the overall number of GPUs involved are the same (2 GPUs, 3 GPUs, 4 GPUs), there is a similar trend: the configurations which involve local exclusive (Le) GPUs are always the slowest, and the performance increases as they are replaced by shared GPUs and/or the number of these increases.

Despite these observations, suggesting one should always maximize the use of shared GPUs, it cannot be dismissed that, from the individual standpoint of a workload, it is always best to use exclusive GPUs. This was already shown in Figure 4, where the absolute best individual times (60s) where obtained when using local exclusive (Le) GPUs. This motivated the development of the HaaS migration mechanism, by which workloads may be moved from one HaaS backend node to another, to gain local access to (faster) exclusive GPUs.

F. Execution Time with Migration

In the final set of tests, both VM1 and VM2 ran in profile B, meaning workloads can now be initially created in any of these two HaaS backend nodes, and may migrate between them.

One reason to launch workloads in specific backend nodes is to allow them to initially grab a specific exclusive GPU, once such GPUs must be local to the workloads. But HaaS also allows a workload to change its exclusive GPU (or add one if it doesn't have any), which may involve moving the workload to another node (the one hosting that exclusive GPU), all without losing access to the shared GPUs the workload may have been using (shared GPUs that were local become remote, and vice versa, with rOpenCL supporting transparent access to all).

To assess the impact on execution times of workloads migrations, all possible scenarios prone to migration were tested, with the same 3 homogeneous workloads of the previous tests. These scenarios assume that all 3 workloads request an exclusive GPU, and so each scenario provides at least 1 exclusive GPU, up to a maximum of 3, once each workload

TABLE II
MIGRATION-PRONE GPU CONFIGURATIONS WITH 2 HAAS SERVICES.

Locality			*VM2*				
	Mode		**shared**		**exclusive**		**shared & exclusive**
		GPUs	*1*	*2*	*1*	*2*	*2*
VM1	**shared**	*1*	-	-	(1s+1e)	**1s+2e**	**1s+1e1s**
		2	-	-	(2s+1e)	(2s+2e)	**2s+1e1s**
	exclusive	*1*	**1e+1s**	**1e+2s**	**1e+1e**	**1e+2e**	**1e+1e1s**
		2	(2e+1s)	**2e+2s**	(2e + 1e)	-	**2e+1e1s**
	shared & exclusive	*2*	(1e1s+1s)	(1e1s+2s)	(1e1s+1e)	(1e1s+2e)	**1e1s+1e1s**

cannot use more than 1 exclusive GPU. If the exclusive GPU requested by a workload is available, or if there are shared GPUs, the workload starts immediately; if the exclusive GPU requested only becomes available later, the workload will be migrated (if needed) to the hosting node of that GPU.

The admissible scenarios are denoted in bold in Table II (the dual equivalent scenarios are within parenthesis). The notation adopted is similar to the one used previously, but the location (L=Local, R=Remote) of the GPUs is omitted, as it is only relevant if they are shared and/or exclusive, and if they are hosted on the same or different backend nodes (VM1 or VM2).

Even though there is some unpredictability in the initial placement of workloads (which determines some variability in the execution times measured), the expected behaviour is governed by some simple rules and reasonable expectations: a) if a scenario provides n exclusive GPUs (with $n \leq 3$), then n workloads will be initially created on the hosting nodes of those n GPUs, and the remaining $3 - n$ workloads will reserve one exclusive GPU from those n GPUs; b) the initial placement of workloads that couldn't get an exclusive GPU is arbitrary; however, it is expected that Docker Swarm spreads them uniformly within the set of HaaS backend nodes (starting by the nodes with none workload); this means that a workload may, or may not need to migrate when later is assigned the exclusive GPU it has requested; c) all workloads will be served by all shared GPUs provided by the scenario; these shared GPUs may be local and/or remote, depending on the initial placement of the workload and any subsequent migration(s).

Figure 7 shows the results (averages of the best 3 runs with the 3 workloads) of all the scenarios involving migrations.

For each number of GPUs utilized (2, 3, or 4), the fastest scenarios are the ones that maximize the number of exclusive GPUs: 1e+1e for 2 GPUs (125s), 2e+1e for 3 GPUs (64s), and 2e+1e1s for 4 GPUs (62s). This is inline with the comments at the end of section IV-E3, pointing to the expected benefits that could be introduced by the migration mechanism, which allows workloads to become local to their exclusive GPU.

Comparing these results with the best ones obtained so far with the same number of GPUs but without migration support (though they also use 2 HaaS backend nodes), these are the corresponding speedups: 90s (2Ls, Figure 4) / 125s (1e+1e, Figure 7) = 0.72 with 2 GPUs; 78s (2Ls+1Rs, Figure 6) / 64s (2e+1e, Figure 7) = 1.22 with 3 GPUs; 71s (2Ls+2Rs, Figure 6) / 62s (2e+1e1s, Figure 7) = 1.15 with 4 GPUs.

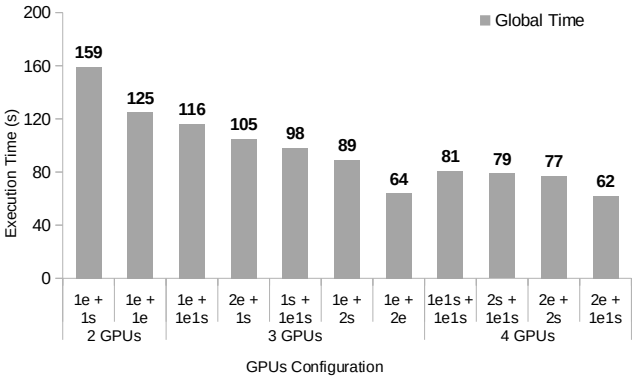


Fig. 7. Execution times with migrations involved (2 HaaS services).

So, with 2 GPUs available, there's no point using more than one HaaS backend node and activate migration. But, with 3 or 4 GPUs, it proved beneficial to scatter the exclusive GPUs and activate migration. And even though the speedups were modest, and the scalability seems limited, it is expected that there is room for performance improvements with more back-end nodes and GPUs, as the number of workloads increase, though further experiments are needed to confirm it.

It can also be seen in Figure 7 that after maximizing the number of exclusive GPUs allowed by the testbed constraints, the performance still benefits from the inclusion of shared GPUs: 2e+1e1s is slightly faster than 2e+1e, with 2e+1e1s being the absolute fastest scenario identified in this paper.

It is also important to understand and justify why some scenarios are faster than others (even though it is not possible to cover all of the situations here). For instance why is 1e+1s (159s) slower than 1e+1e (125s)? In 1e+1e there will be 1 workload on VM1 and another on VM2, each using an exclusive GPU, while the 3rd workload will be waiting for one of the GPUs to become free; with 2 workloads executing (and finishing) at the same time, followed by the 3rd workload, the global execution time will roughly be twice the individual execution time of one workload when using an exclusive GPU (individual time of scenario 1Le in Figure 4), that is, 2×60s = 120s $\lessapprox$ 125s. On the other hand, the execution time of 1e+1s will be higher (159s) because only one workload at a time will use the single exclusive GPU and most probably that will imply migration for at least one (if not the two) of the other workloads; also, all workloads will use the single shared GPU, with at least one of them using it remotely (the workload currently using the exclusive GPU), but possibly more.

Consider yet another example: what happens if 1e+1s (159s) gains an extra shared GPU, unfolding into 1e+2s (89s)? Then, the global execution time basically matches the one of 2Ls (90s, see Figure 4): one workload will heavily use an exclusive local GPU, like in a 1Le scenario, and will very lightly use the 2 remote shared GPUs; with high probability, the other two workloads will be local to the shared GPU, in which case they will share it very efficiently, as in a 2Ls scenario, maybe not needing to ever use the exclusive GPU; this means that a global 2Ls time (90s, see Figure 4) is overlapping and encompassing

and individual 1Le time (60s, see Figure 4), meaning they all end after $90s$ ($\approx$ the 89s measured for 1e+2s).

Finally, with at least 3 exclusive GPUs available (scenarios 2e+1e and 2e+1e1s), the global execution times of the 3 workloads (64s and 62s) are very close to the individual execution time of 1 workload with 1 exclusive GPU (60s from 1Le in Figure 4). This is expected, once all workloads are using an exclusive GPU independently of the others.

The results in Figure 7 are also coherent with the ones obtained in the previous tests (Figures 4 and 6) involving the same number and mode (shared vs exclusive) of the GPUs, despite differences in the GPU locality. Again, it is unfeasible to make all comparisons here, and only a few are provided:

- 2 GPUs, 1 exclusive, 1 shared: 1e+1s (159s), is slower than 1Ls1Le (153s, Figure 4), once in 1Ls1Le the GPU shared is always local for all workloads, while in 1e+1s it may not be; the same reasoning may be applied to justify why 1e+1s is faster than 1Le+1Rs (180s, Figure 6);
- 2 GPUs, 2 exclusive: 1e+1e (125s) has a similar time to 2Le (123s, Figure 4); this is because in 1e+1e, the workload awaiting for an exclusive GPU won't be migrated; the container for that workload is not really created until the exclusive GPU it requested becomes free, and then the workload will be created in the node of that GPU;
- 3 GPUs, 2 exclusive, 1 shared: 1e+1e1s (116s) and 2e+1s (105s) are (much) faster than 2Le+1Rs (160s, Figure 6), since in the latter the shared GPU will always be remote for all workloads, while in the formers it may not be.

V. Conclusion

HaaS is a distributed platform that offers Hashcat as a containerized service able to harnesses the computing power of local and remote GPUs to accelerate password cracking. It was build on top of rOpenCL, which provides a unified view of the distributed GPUs. HaaS includes services for users, GPUs and workloads management, supporting different GPU sharing modes, GPU reservations and workloads migration, for increased GPU usage efficiency and workload performance.

The tests carried out allowed to validate the architectural decisions made, showing their impact on usability and performance. They proved the ability of HaaS to adapt to the specific needs of each user and to the runtime conditions, and confirmed the robustness of the current implementation in dealing with different GPU configurations and multiple workloads. But, despite the observed improvements brought by their addition, the tests have also shown the inherent performance and scalability limitations of using remote GPUs.

We intend to improve HaaS in several ways, namely: to allow users to assign their workloads more than one exclusive GPU and also subsets of shared GPUs; to add mechanisms to monitor the load and memory of the GPUs in order to prevent the oversubscribing of shared GPUs (which will also require a new reservation mechanism for shared GPUs). Further tests, with an heterogeneous mix of GPUs, will also allow to better understand the tradeoffs between GPU locality and performance. A comparison with Hashtopolis is also due, ideally in a cloud setting and with GPUs with more RAM (which would enable more than three simultaneous workloads).

HaaS, with its innovative architecture and features, has the potential to become a valuable tool for security professionals and researchers, helping to protect data and systems from cyber threats. HaaS Docker Hub images are available online [20], though they currently should be considered experimental and need further refinement to be considered production-level.

References

[1] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano, "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 553–567.

[2] B. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz, "Passgan: A deep learning approach for password guessing," 2019. [Online]. Available: https://arxiv.org/abs/1709.00440

[3] D. Apostal, K. Foerster, A. Chatterjee, and T. Desell, "Password recovery using mpi and cuda," in *Proceedings of the 2012 19th International Conference on High Performance Computing*, 2012, pp. 1–9.

[4] J. Steube, "Hashcat: Advanced password recovery." [Online]. Available: https://hashcat.net/hashcat/

[5] K. Hamidouche, A. Venkatesh, A. A. Awan, H. Subramoni, C.-H. Chu, and D. K. Panda, "Exploiting gpudirect rdma in designing high performance openshmem for nvidia gpu clusters," in *Proceedings of the 2015 IEEE Intern. Conference on Cluster Computing*, 2015, pp. 78–87.

[6] R. Alves and J. Rufino, "Extending heterogeneous applications to remote co-processors with ropencl," in *Proceedings of the 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 305–312.

[7] A. Narayanan and V. Shmatikov, "Fast dictionary attacks on passwords using time-space tradeoff," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 364–372.

[8] M. Weir, S. Aggarwal, B. d. Medeiros, and B. Glodek, "Password cracking using probabilistic context-free grammars," in *Procs. of the 2009 30th IEEE Symp. on Security and Privacy*, 2009, pp. 391–405.

[9] E. Zhou, Y. Peng, G. Shao, F. Deng, Y. Miao, and W. Fan, "Password cracking using chunk similarity," *Future Generation Computer Systems*, vol. 150, pp. 380–394, 2024.

[10] van Hauser, "Thc-hydra." [Online]. Available: https://github.com/vanhauser-thc/thc-hydra

[11] T. D'Otreppe, "Aircrack-ng," https://www.aircrack-ng.org. [Online]. Available: https://www.aircrack-ng.org

[12] F. Chantzis, "Ncrack," https://nmap.org/ncrack/. [Online]. Available: https://nmap.org/ncrack/

[13] A. Peslyak, "John the ripper: Password cracker." [Online]. Available: https://www.openwall.com/john/

[14] R. Hranický, M. Holkovič, and P. Matoušek, "On Efficiency of Distributed Password Recovery," *Journal of Digital Forensics, Security and Law*, 2016.

[15] S. Coray. Hashtopolis - github. [Online]. Available: https://github.com/hashtopolis

[16] J. Ulrich. (2023) Unlocking the cloud: Exploring distributed password cracking and the power of hashtopolis. [Online]. Available: https://axelum.eu/en/article-gpu-cloud-password-cracking

[17] R. Alves and J. Rufino, "Leveraging shared accelerators in kubernetes clusters with ropencl," in *Proceedings of the 2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA)*, 2024, pp. 189–192.

[18] J. Solanti, M. Babej, J. Ikkala, and P. Jääskeläinen, "Pocl-r: Distributed opencl runtime for low latency remote offloading," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, 2020.

[19] D. Rachmawati, J. Tarigan, and A. Ginting, "A comparative study of message digest 5 (md5) and sha256 algorithm," in *Journal of Physics: Conference Series*, vol. 978. IOP Publishing, 2018, p. 012116.

[20] C. Lima, "HaaS - Hashcat as a Service (docker hub images)." [Online]. Available: https://hub.docker.com/u/carlossouzal