Digital Investigation 22 (2017) S39–S47

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2017 USA — Proceedings of the Seventeenth Annual DFRWS USA

SCARF: A container-based approach to cloud-scale digital forensic processing

Christopher Stelly*, Vassil Roussev

GNOCIA, Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA

Keywords:
Real-time forensics
Large-scale forensics
Containers
Cloud computing
SCARF

A B S T R A C T

The rapid growth of raw data volume requiring forensic processing has become one of the top concerns of forensic analysts. At present, there are no readily available solutions that provide: a) open and flexible integration of existing forensic tools into a processing pipeline; and b) scale-out architecture that is compatible with common cloud technologies.

Containers, lightweight OS-level virtualized environments, are quickly becoming the preferred architectural unit for building large-scale data processing systems. We present a container-based software framework, *SCARF*, which applies this approach to forensic computations. Our prototype demonstrates its practicality by providing low-cost integration of both custom code and a variety of third-party tools via simple data interfaces. The resulting system fits well with the data parallel nature of most forensic tasks, which tend to have few dependencies that limit parallel execution.

Our experimental evaluation shows that for several types of processing tasks—such as hashing, indexing and bulk processing—performance scales almost linearly with the addition of hardware resources. We show that the software engineering effort to integrate new tools is quite modest, and all the critical task scheduling and resource allocation are automatically managed by the container orchestration runtime—*Docker Swarm*, or similar.

© 2017 The Author(s). Published by Elsevier Ltd. on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Introduction

The problem of maintaining constant case turn-around times in the face of continued growth in data volume, first discussed more than ten years ago (Roussev and Richard, 2004; Richard and Roussev, 2006), has been steadily climbing up the list of top-level concerns for digital forensic investigators, as case overload has lead to the growth in backlogs.

In 2016, a US Department of Justice audit of the New Jersey Regional Computer Forensic Laboratory (RCFL) (Department of Justic, 2016) found that there were 194 service requests that were not closed within 60 days, including 39 that were more than a year old. Every month between January and June of 2015, the fraction of cases open for at least a year varied between 17 and 22%. An earlier DOJ Audit Report (U.S. Department of Justice, 2015) shows that, across FBI's 16 RCFL units, there were 1566 open requests as of August 2014. Of these, almost 60% were over 90 days outstanding: 381 (24.3%) were between 91 and 180 days old, 290 (18.5%)—between 6 and 12 months, and 262 (16.7%) were over a year old.

Conceptually, the problem is the result of three core trends that shows no signs of abating: a) the size and complexity of forensic targets will continue to grow at an exponential rate for the foreseeable future; b) human resources charged with the problem will *not* grow appreciably, relative to the expected data growth; c) there are real-world deadlines to any digital forensic analysis, so turn-around time must remain stable, despite the expected data growth and growing data load *per* forensic expert.

To some degree, the performance inadequacy of current forensic systems has been masked by stagnant growth in throughput performance of large capacity HDD. For example, a modern 10 TB drive (Seagate Technology, 2016) provides only about 250 MB/s of sustained transfer rate, requiring over 11 h (under optimal conditions) to fully read its data content.

However, advances in SSD storage – fast-falling costs, and rapid capacity and bandwidth growth – are quickly reshaping the storage technology environment. The introduction of NVM Express Workgroup (2016) devices pushes the device-to-host throughput rates above 1 GB/s; a number of consumer laptops are already equipped with integrated flash storage, providing transfer rates of 1.6 GB/s and up. Notably, such devices are equipped with

* Corresponding author.
E-mail addresses: cd@stelly.org (C. Stelly), vassil@roussev.net (V. Roussev).

http://dx.doi.org/10.1016/j.diin.2017.06.008
1742-2876/© 2017 The Author(s). Published by Elsevier Ltd. on behalf of DFRWS. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Thunderbolt 2/3 external interfaces providing up to 40 Gb/s of outgoing bandwidth. In effect, this removes device I/O as a throughput bottleneck for flash memory acquisition.

Networking advances have made 10/40 Gb Ethernet a commodity technology, and even low-latency 100 Gb *InfiniBand* network switches (typical of high-performance computing environments) have become practical for forensic lab deployment with per-port cost of $400.

In sum, the technology landscape has changed *qualitatively*: acquiring the data and transferring for processing over the network (to a cluster of compute nodes) is no longer the performance bottleneck; CPU processing is. Thus, it is critical to develop scale out solutions that can utilize the dramatic increases in commodity I/O rates and match them with a corresponding increase in processing rates.

We are interested in building an *open* and *scalable* platform that provides a set core services out of the box and allows for seamless integration of third party at minimal development cost. From a legal and scientific perspective, the original argument for having open forensic tools was made, at length, by Carrier (2002); in short, open tools support a much greater level of transparency and reproducibility, resulting in increased trust of the process. Importantly, the frequent use of open tools leads to lower error rates, as the implementation errors are regularly exposed and repaired.

Individual open (source) forensic solutions provide substantial analytic capabilities. However, they tend to be specialized and are developed independently (based on driving scenarios) with very limited, if any, support for integration. This means that there is an implied additional cost to integrate and operate these tools as part of a comprehensive forensic solution. The few existing integrated environments, of which TSK (Carrier) is the most popular, are not designed from the ground up for scale out performance. Experience shows that it is difficult to retrofit such fundamental property into a system; one example of this is the short-lived effort to extend *Sleuthkit* with *Hadoop*'s big data processing capabilities (Stewart, 2012).

The ability to *easily* integrate a growing set of new capabilities implemented in a variety of language platforms. TSK follows the traditional model of tight, compile-time API-based integration. While such a model has some advantages, such as minimal communication overhead, it is only sustainable by popular-enough projects where a large group of contributors builds and maintains language bindings for the most commonly used languages; unfortunately, TSK is not in this category. Other projects, like (Volatility Foundation (2007–2015) and The Rekall Team, have confined themselves to a single language (*Python*); however, they only deal with a single (if important) sub-domain – memory analysis. Also, users must learn to work around the substantial performance deficiencies of the *Python* language, which can be two-three orders of magnitude slower than a *C*-based solution (Richard and Case, 2014).

Our basic thesis is that an *open* and *comprehensive* digital forensic environment should be language-agnostic and should rely only on simple data interfaces and a shared evidence database as a means to integrate the functionality of different modules into a complete solution. Further, we put forward the idea that *containers*, a lightweight virtualization solution that is quickly gaining popularity as an alternative to full-stack virtualization, is a particularly good fit for the needs of a scalable and extensible digital forensic environment.

We describe our experiences in building a working prototype of such a system, and present an experimental evaluation that strongly supports our thesis both with respect the ability to integrate a disparate set of forensic tools into a unified processing pipeline, and the ability to scale out such a system to the available hardware.

Related work

As already discussed, a dozen years after the recognition that data volume growth is on course to overwhelm the typical hardware and software platforms used by digital forensic analysts, the problem has risen to a top-level *practical* concern over the last 3–4 years.

Put together, these observations point to the need to develop forensic processing systems that can scale out in order to match the growing demand with a comparable increase in hardware resources. Following the discussion in (Roussev, 2011), we refer to this property as *data scalability* and there are hardly any systems that provide a complete solutions. The *Hansken* project at the *Netherlands Forensic Institute* (Institute, 2016; van Beek et al., 2015) comes closest to fulfilling this vision on the basis of commodity big data technology. Although it looks promising, it is not open source and it is not possible to independently evaluate its performance. (Previous work (Roussev et al., 2009) has shown that early versions of *Hadoop* provided little benefit due to the high latency induced by disk access between the two phase of *map*/*reduce* processing.)

Garfinkel's *bulk_extractor* tool (Garfinkel, 2013) performs stream-oriented processing and data extraction from forensic images. It is one of the few robust solutions that explicitly targets the handling of large data sets. Optimizations, such as compiled regular expressions and built-in multi-threaded processing, allow it to reach throughput of 60 MB/s (in default configuration) using 48 cores (Roussev et al., 2013). The two main shortcomings here are the lack of support for file-based processing and the lack of support for cluster processing, which would be particularly appropriate.

The ultimate measure of a solution's data scalability is maintaining constant processing times in the face of ever growing processing demands. In other words, forensic processing exhibits (soft) real-time properties (Roussev et al., 2013), which can be expressed in terms of throughput rates.

Extensibility—the ability to incrementally accommodate new functional modules into the forensic flow at near-zero cost—is also an important component of the needed solution, as it enables the analysis of new types of evidence to match the increasing diversity of forensic targets. At present, extensibility is limited to tight, API-based software integration of additional modules (plug-ins).

For tools that are designed as specialized modules, such as *ExifTool* (Harvey) from the outset, it is important to support *composability*—the ability to *easily* incorporating the module as part of a larger infrastructure. At present, composability support is usually limited to a bespoke output format specification.

Containers

Although containers have only recently gained popularity as a core IT management mechanism, the original idea of providing constrained environments for (groups of) processes to execute in can be traced back to the chroot *Unix* command. It restricts the file namespace visible to a process and was introduced into the 4.2BSD distribution by Bill Joy (Kamp and Watson, 2000) (apparently for the purposes of building and testing new versions of the code). It was later developed into the more comprehensive *jail* mechanism in FreeBSD 4.0 (Kamp and Watson, 2000), which provides more elaborate containment mechanisms. “Jails are typically set up using one of two philosophies: either to constrain a specific application (possibly running with privilege), or to create a *virtual system image* running a variety of daemons and services.” (Kamp and Watson, 2000) These remain the two basic usage scenarios to this day.

The *VServer* project (des Ligneris, 2005) marked the first step in the adoption on *Linux*, which allowed the "running several general purpose *Linux* server on a single box with a high degree of Independence and security."[1] It was an influential development, which was adopted widely by ISPs to provide *Virtual Private Servers*. *OpenVZ*, based on the *Virtuozzo* project, was another parallel effort, which was open sourced in 2005 (OpenVZ Virtuozzo Containers Wiki, 2016). In either case, the main technical impediment to a wider adoption was the need to patch and rebuild the kernel.

Menage (2007) introduced the term "generic process containers", the full vision of which took several years to implement. The initial steps of the implementation became known as cgroups (control groups) as part of Linux 2.6.24 (Corbet, 2007). The last major component needed were *user namespaces*, which allow per-process namespaces; they provide basic means to limit the visibility (and access) to resources, such as mount points, PID numbers, and network stack state. User namespaces became part of Linux 3.8 (2013) and, combined with userland tooling developed by the LXC (2008–17) project, provided the first out-of-the-box container deployment and management facilities.

Container management and orchestration

The standardization of kernel mechanisms needed for container operation opened up the opportunity for multiple userspace tools to be developed in addition to *LXC*. Among these, *Docker* (docker.com) (Merkel, 2014) is the most popular one; however, several other projects – Google's *Kubernetes* (kubernetes.io) (Burns et al., 2016), *rkt* (coreos.com/rkt), and *LXD* (linuxcontainers.org/lxd/) – also have strong industry backing. This has lead to a quick maturation of the technology and an active effort, the *Open Container Initiative* (opencontainers.org), to standardize both the image format and the runtime interface. These standards guarantee interoperability and set up a best-of-breed competition among the tools.

For the purposes of our experimental work, we have chosen *Docker* as the prototype implementation platform. However, the solution could easily be ported to a different container management environment with minimal effort. Indeed, our experience shows that a more sophisticated load balancing platform would have saved us non-trivial amounts of effort.

The core *Docker* platform provides the means to manage the full lifecycle of a container–creation, updates, persistent storage and versioning, instantiation, resource provisioning, and shutdown. As already mentioned, a container consists of a group of (one or more) processes, along with access to a set of resources–CPU cores, RAM allocation, file systems, and networking–needed to perform the computational task. All containers share a common OS kernel but, by default, are isolated from each other; it is also possible to setup sharing of resources where needed, e.g., software installations. To running processes, there is no apparent difference between executing in a container versus running in a full-stack virtual machine (with a dedicated kernel). The main benefit is that a container takes much fewer resources (than a VM) and the cost to instantiate/shutdown it down is much lower–comparable to that of a regular process.

In sum, the modern container is an evolved version of the original *process* concept with a higher degree of autonomy, portability, and manageability. Container instances are encapsulated units of schedulable work, complete with all the necessary resources–code, data, and runtime configuration.

[1] http://www.solucorp.qc.ca/changes.hc?projet=vserver&version=all.

Cluster management and orchestration

Recall that our ultimate goal is to effectively employ a compute cluster to perform the forensic processing at maximum I/O rates. This means that we need an infrastructure that can manage and schedule the available resources. For this project, we use the *Docker Swarm* facility, which was initiated as a standalone project, but has recently been incorporated as a core function into *Docker Engine*; we refer to it as *swarm*.

A swarm is a cluster of *Docker* engines, or *nodes*, on which services are deployed. The Docker Engine CLI and API include commands to manage swarm nodes, such as to add and remove nodes, and deploy and orchestrate (schedule) services across the cluster.

The deployment of services is performed by *manager* nodes, which dispatch units of work, *tasks*, to *worker* nodes. Manager nodes manage and orchestrate the cluster such that the desired state of the swarm is maintained; for that purpose, they elect a *leader* to conduct the orchestration. *Worker* nodes receive and execute tasks dispatched by the managers, and report back on the status of the assigned jobs. This feedback allows the efficient scheduling of the tasks to available resources.

A *service* is the definition of the tasks assigned for execution to workers, and the primary means by which users interact with the swarm. The service definition consists of a container image, which provides the execution environment, and the specific commands to be executed. *Replicated* services allow for a given number of tasks to be distributed across the cluster; *global* services run on every node of the swarm.

Docker's swarm manager uses ingress load balancing to expose the services provided by the swarm via a given network port. The swarm has an internal DNS component that automatically assigns each service in the swarm a DNS entry. The manager distributes requests among the services within the cluster based upon the DNS name of the service. Once dispatched, a task cannot migrate–it must run to completion, or fail.

In sum, *Docker* cluster orchestration provides the necessary *basic* functions to scale out a computation across a sizeable cluster; however, its performance optimization techniques are relatively simple (at this stage) and do require some effort on part of the developer to achieve good performance.

SCARF

In this section, we describe the design and implementation of our prototype solution called *SCARF* (SCAlable Realtime Forensics). It is an experimental system that allows us to evaluate the costs, benefits, and limitations of a container-based approach to building an integrated forensic platform.

We define *containerization* as the encapsulation of individual executable modules as fully autonomous images that require only a base OS installation to execute. Images are instantiated as individual *tasks*, which are the basic units of scheduled CPU work. Tasks run as individual processes, or groups of processes, that perform a particular function and execute in a constrained environment.

We expect containerization to bring two main benefits: a) scale out performance for *data parallel* operations, such as any *per-file* computations like hashing and data extraction; b) low-cost extensibility with new functional modules.

Data ingest

For our experiments, we used NTFS images, which we preprocess before initiating the actual data runs. Reading and parsing the NTFS header information takes a near-negligible amount of time (Table 1). After ingesting the header information,

Table 1
NTFS metadata retrieval times.

Number of files	NTFS parsing time
56,274	1.5 s
619,555	29.8 s

it is stored in memory for the duration of the process lifetime.

Once image metadata has been obtained, it is possible to optimize file acquisition. The optimization can vary on source type, but commonly requires a linear, or striped, reading of the media. To accomplish a linear read, an inverse mapping is built between logical blocks and file identifiers. Assuming that neighboring blocks are laid out in physical proximity (still a reasonable assumption for HDDs), the read stage can then begin in order of physical presence on disk, allowing for a linear read for contiguous files.

For fragmented files, the process is somewhat more complicated. A linear read across a fragmented file would necessarily read blocks belonging to other files. To avoid distributing extra data and data client-side file reassembly, we reconstruct the files in RAM as they are read, and only stream them out to processors when the file is entirely in memory. Although it is entirely possible that some files may exceed available RAM, this was not really a concern for our machines with 256 GB of RAM. Nonetheless, by default, we handle large files by streaming their content to the node straight from disk.

Architecture

Fig. 1 provides an overview of the functional components and main data flows of SCARF. The *data broker* extracts the raw data from the forensic target and prepares it for streaming to the cluster nodes. The broker serves as an abstraction layer that decouples the processing of the data from its source format, such as a filesystem, RAM snapshot, or network capture. At present, we support two type types data access: *bulk streamer* and *file streamer*.

The *bulk streamer* provides sequential block-level access to an entire volume without attempting logical artifact reconstruction; it is suitable for tools like *bulk_extractor* that function at the same level of abstraction and look for relatively small pieces of data.

The *file streamer* reconstructs files from block storage and transmits them as units of input data. This is based on reading the filesystem data structures (during initialization) and reconstituting the files on the fly; we utilize a version of the *LOTA* approach described in (Roussev et al., 2013) to optimize access times.

Task manager

The next major component is the *task manager*, which keeps persistent logs of task definitions and task completions. The definitions are generated by the data broker and depend on the set of available functions and the stream of data extracted from the target. As the simplest example, for every file identified on the target (by parsing the filesystem metadata) the broker will generate a *SHA1* task, which will placed in the task definition log; we refer to it as the *task queue*.

The task manager uses *Apache Kafka* (kafka.apache.org) to maintain its persistent logs and notify registered *data client* nodes of available tasks. Conversely, completed tasks are committed to the completed log. Although not a central point of this paper, we should emphasize that maintaining reliable logs of completed processing is *critical* to ensuring the integrity of the

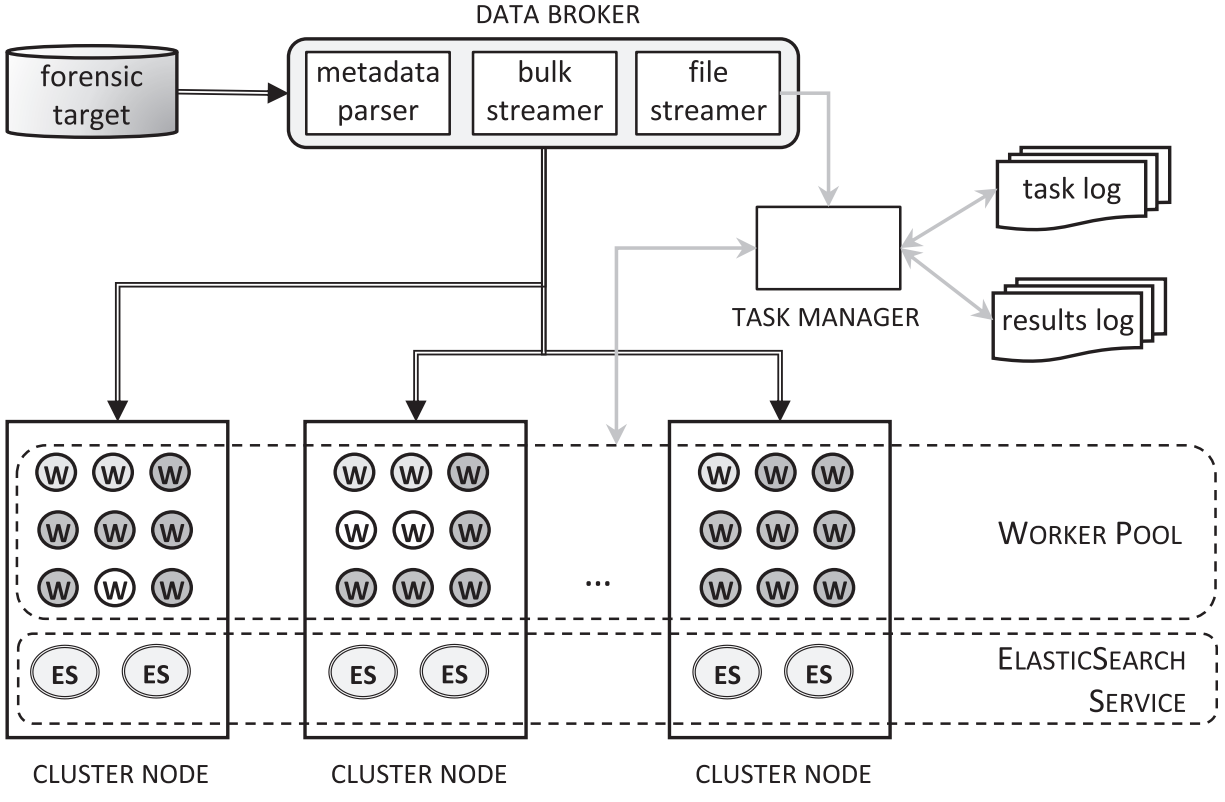

Fig. 1. Architectural sketch of SCARF (simplified).

computation. At scale, errors in complex distributed systems happen with some regularity; experience shows that it is infeasible to eliminate all possible source of failures. For example, in our case, out of the millions of spawned container instances, some will fail to execute.

The practical way to handle sporadic failures is to restart the computation; if the failure continues to occur, then it is systematic and needs to be debugged. We should note that failure can take the form of a task taking too long to execute, in which case it is better to terminate and restart it. Google's experience with map-reduce processing (Dean and Ghemawat, 2008) shows that a small fraction of the tasks tend to delay the overall completion of the processing. The solution to this is to run multiple instances of the same task, and only take the result from the first one to finish.

The main point is that keeping reliable detailed logs of the processing is absolutely necessary for both integrity and performance reasons.

Data clients

A *data client* is a container instance that receives a portion of the forensic target and organizes execution of *tasks* on the given portion. Ultimately, the entire forensic target is distributed across a set of data clients with the aim of optimizing the load across the nodes.

Upon creation, data clients generate a unique identifier and register to both the data broker and the task manager. As the data broker streams data from a forensic target to data clients, it simultaneously polls the task manager for outstanding tasks.

Tasks are added to the task manager with the unique identifier of the data client and can optionally be accompanied by file identifiers (in the case of a file streamer). Upon receipt, the data client becomes responsible for the execution of a given task. Further, a *future* option can be added so that subsequently received files will also have that task executed upon it. This feature allows tasks to return results *even as data is streamed from the original source*.

Task execution is *not* handled by the data client - it is farmed out to a dynamic pool of specialized containers, called *workers*. The rationale here is that separation of duties between storing data in memory and execution of forensic tasks upon the data are fundamentally different tasks. Further, separating worker containers from data clients allows for easy development and deployment of new types of workers.

Workers

A *worker* is a container instance that performs a specific task on given piece of input data. Importantly, workers have no concept of files – they simply provide a remote procedure call (RPC) interface, through which they consumes incoming data and produce a result in the form of a JSON string. Although the interface would benefit from the imposition of some additional structure on the I/O stream formats, this generic approach is quite flexible and allows easy creation of new types of workers.

The containerization of workers offers the ability to easily scale any task. This can be extended to easily *prioritize* tasks. That is, we can develop methods to automatically scale containers based on server availability or task importance. Within Docker, prioritization is easily implemented with the *docker service scale* command. (We would expect more complex algorithms to eventually coordinate scaling as part of the underlying orchestration service.)

As an illustration, the following will increase the number of containers running an *ExifTool* worker:

```
docker service scale exiftools=48
```

By consistently applying the above RPC approach, it is possible to scale out any container, regardless of the specific computation performed.

Data clients are not aware of the number of, or network paths to, workers. Instead, network resolution (and rudimentary load balancing) is mediated by an internal DNS service. Thus, a lookup for an 'exiftools' container will return a virtual IP to the least recently used container.

For our initial testing, we have implemented a variety of workers representing workloads with different computational demands:

- *SHA1*: performs crypto hashing of data;
- *grep*: live regular expression search;
- *Tika*: text extraction with *Apache Tika*;
- *open_nsfw*: image classification using a trained *Caffe* deep neural network provided by Yahoo![2];
- *bulk_extractor*: feature extraction using regular expressions and verification (Garfinkel, 2013);
- *ExifTool*: metadata extraction from a file.

Results repository: ElasticSearch

As tasks complete, results are returned to the data client. After a batch of results is returned, or a predefined space threshold is reached, the results are bundled and sent to a cluster of *ElasticSearch* (ES) nodes.

The ES cluster nodes are deployed on the same hardware as the workers and provide a searchable database of the results from the tasks. It is integrated via its RESTful interface for storage, querying, and retrieval of data, and can dynamically scale to meet the incoming stream of results.

The ES modules are split between gateway, data, and master nodes. *Apache Tika* is also deployed alongside to enable the indexing of non-plaintext MIME types, such as PDF documents.

Container management

The coordination of deployment of containers across a set of worker nodes is performed by a background service running on all the hardware nodes. The service is responsible for adding more containers as existing containers become overwhelmed with work. It is also tracks container failures, and optionally, can automatically start replacement ones. Finally, it provides a mechanism for service discovery as additional services are brought online. For our prototype, we use *Docker Swarm* to provide as the core of all container management-related functions.

Extending SCARF

One of the design goals of SCARF is to allow for the easy addition of additional *Worker* types. That is, *extensibility* is a primary concern.

Containers, here treated as individual and distinct computational units, provide a perfect platform for extensibility. Namely, they can be scaled (launched and retired) according to available resources or priority. Importantly, *no modification* is required to an existing tool in order to accomplish scalability. Further, *any* forensic tool can deployed within a container, and thus to SCARF – as discussed previously, these workers simply take in data, perform an operation, and return a JSON string.

For most single-purpose tools, the process of incorporation into SCARF's processing fabric consists of three basic steps.

[2] https://github.com/yahoo/open_nsfw.

Build an RPC wrapper

First, we apply a small amount of wrapper code around the tool. The goal is to expose a network path for the acquisition of forensic target data as well as to provide a response mechanism. The wrapper provides the means to execute the existing tool on the provided input data and to return the results back to SCARF. This can be accomplished in a variety of ways; we have chosen to employ *golang*'s excellent built-in RPC library. Using RPC allows for the results of a forensic operation to be returned while simultaneously providing a mechanism to acquire the forensic target.

Once an RPC function is invoked from SCARF, the given input is translated into a form that the tool can operate on, and the necessary command-line parameters are generated to launch the tool. In the developed examples, this involved changing two lines of code as *all* RPC, logging, and error checking code can be templated and abstracted away.

For example, the wrapper for *ExifTool* consists of the following code:

Listing 1. ExifTool RPC wrapper code

```
func (t *RPC) Execute(args *Args, reply *string) error {
    toolPath := "/usr/bin/exiftool"

    // Setup the shell command to launch ExifTool
    // - indicates data will be read from STDIN
    opts := []string{"-"}

    cmd := exec.Command(toolPath, opts)
    cmd.Stdin = bytes.NewReader(args.Data)
    var out bytes.Buffer
    cmd.Stdout = &out

    err := cmd.Run()
    fmt.Println(out.String())
    *reply = out.String()
    return err
}
```

The overwhelming majority of the code in Listing 1 is quite generic and is useful as a template for the wrapping of similar tools. The two key variables are *toolPath* and *opts*, both of which will be dependent on the forensic tool in question.

Containerize the tool

Second, the tool and wrapper must be implemented into a container image, a process often referred to as *containerization*. This requires the development of a container description file; in the case of Docker, it is called *Dockerfile* and provides simple script for building the image. The starting point is a known system image, such as a clean operating system installation. The series of steps includes operations such as installation of prerequisite software and setting the environment variables. The wrapper binary is also installed and set to auto-execute upon container creation. Listing 2 (wrapped to fit the column) shows a *Dockerfile* to build a *bulk_extractor* image.

The majority of commands to build the tool should be familiar to open source developers. The installation of pre-compiled binaries would be clearly easier; however, in cases where building from source is desired, pre-existing build sequences can be added to a Dockerfile with minimal effort.

Listing 2. Dockerfile for *bulk_extractor*

```
FROM ubuntu:trusty
MAINTAINER joe <joe@example.com>

RUN apt-get update
RUN apt-get install -y curl make g++ gcc \
                netcat dnsutils vim flex \
                libewf-dev libssl-dev wget

RUN wget http://digitalcorpora.org/downloads/bulk_ex
    tractor/bulk_extractor-1.5.5.tar.gz
RUN tar xvzf bulk_extractor-1.5.5.tar.gz
RUN cd bulk_extractor-1.5.5/ &&
        ./configure && make &&
        sudo make install

ADD bin/rpcserver /
ADD banner.txt /

RUN mkdir -p /tmp/bulk_in/
RUN mkdir -p /tmp/bulk_out/

CMD [''/rpcserver'']
```

Invoke the tool

Once the tool is containerized and exported as a network service, it is ready to be incorporated into the processing fabric. This is accomplished by adding a few lines of code within the *consumer*. The code to invoke *ExifTool* is shown on Listing 3:

Listing 3. *ExifTool* invocation

```
exifclient, _ := rpc.DialHTTP("tcp", *exifServer)
args := &rpcshared.Args{DataID: string(fileId),
                        Data: fileData[:]}
var reply string
rpcerr := exifclient.Call("RPC.Execute", args, &reply)
checkError(rpcerr)
```

Performance evaluation

This effectiveness of the proposed framework relies on the ability to scale operations. In order to demonstrate scalability, we have selected several tools to benchmark common forensic operations under a variety of conditions. Although a side effect these examples results in a processing rate that can (in some cases) keep up with SATA speeds, the important factor is the relationship between throughput and the number of deployed containers. We show that an increased number of containers results in an increase of throughput, thus additional hardware could be deployed, resulting in more containers, which results in higher throughput.

Processing rates

The base configuration of our evaluation setup consists of a cluster of four rack-mounted server machines connected to a commodity 10 GbE switch. Each box has 256 GB RAM, 24 2.6 GHz

dual-threaded cores for a total of 96 physical cores and 192 logical ones. All nodes have a SATA-attached 1 TB SSD (Samsung 850 Pro), although this is largely irrelevant as all data for processing is handled in RAM. The observed throughput for bulk transfer over TCP connection was about 1 GB/s.

At five years, the CPUs are three generations old and near the end of their lifecycle. The upside is that the results can be considered more representative as the hardware would be easily affordable for any lab.

The reference test data is a full 200 GB NTFS image, which was created by using a random selection of files from the *GovDocs* corpus (Garfinkel et al., 2009).

For benchmarking purposes and ease of analysis, we limit each container to a single CPU core. Further, we consider the processing functions one at a time in order to understand their intrinsic performance characteristics. The times shown are inclusive of all overhead, including network communications among the active containers.

Crypto hashing

Cryptographic hashing is a common forensic function. For this processing scenario, we have selected the SHA-1 hashing algorithm. We have developed a container which provides a remote procedure call over TCP using the methods outlined in the previous section. The container returns the hash of given data.

As the results in Table 2 show, as few as 12 containers almost saturate the available network bandwidth of 1 GB/s.

Metadata extraction

The *ExifTool* (Harvey) is commonly deployed tool used to extract metadata from file content; it supports a large number of file formats and attributes. Listing 1 shows the RPC wrapper code that – along with the *ExifTool* v10.10 executable are placed in a container image.

Considering the end points of the experimental space (Table 3) – 4 and 192 containers, respectively – we observe near-linear speedup from 5 to 192 MB/s. This is in line with expectations as the metadata extraction does not depend on I/O and the workload is inherently data parallel. Considering the whole range of parameters, we can see that the average throughput per container follows a bell curve distribution with a sweat spot at 32 containers.

Image classification

Yahoo! recently released an open source image classifier, OpenNSFW (github.com/yahoo/open_nsfw). This is a deep neural network, built on top of Caffe (caffe.berkeleyvision.org), which comes pre-trained to detect pornographic images. For every image that is processed, the system yields a value representing confidence in an image's resemblance to pornography.

This is an interesting case as it allows us to assess the cost of providing smarter tools for automated processing that provide results closer in abstraction level to that of the analyst. As Table 4 shows, these are expensive operations and despite the linear scaling with respect to the number of files classified, the absolute numbers are much lower than with other tools.

At the same time, compared to the current alternative of a human manually examining (thumbnails of) the images, even this unoptimized solution can classify 138,600 per hour, or 3.33 million over 24 h. By employing more modern CPUs, as well as GPUs, the system can be scaled up to whatever degree is needed to handle lab workloads.

Unlike previous examples, OpenNSFW is *already* provided as a docker container. We only needed to write a small wrapper function to encapsulate the embedded program as an RPC call. As more developers adopt container solutions, such as Docker, integration with SCARF will become even easier.

Indexing common filetypes

Extraction of plaintext data from encoded documents is a common step in forensic analysis. One of the more popular solutions is the *Apache Tika* (tika.apache.org) open-source project. It is specifically designed for this purpose and is often used in conjunction with an indexing engine like *Solr*, or *ElasticSearch*.

From Table 5, it is immediately clear that this workload is much more demanding and only scales sub-linearly. Particularly notable is the drop in processing rate per container between 96 and 192. Recall that there are only 96 physical cores and, for this workload, it appears that the addition of hardware supported threads does not materially improve performance. This suggests that the workload is very effective at utilizing all the CPU's functional units; hence, the addition of threads only marginally improves upon the amount of work being performed.

Bulk Extractor

Bulk extractor (Garfinkel, 2013) is a forensic tool used to analyze raw data streams. Using pre-compiled scanners based on *GNU flex*, it is an effective tool for extracting specific pieces of information, such as emails, URLs, IP addresses, credit card numbers, etc., from a data stream.

The process of wrapping and containerizing the tool was described earlier. What is interesting in this case is that *bulk_extractor* supports multi-threaded execution by default. Therefore, we approached these scenarios in a slightly different manner by exploring the optimal number of CPUs that a *bulk_extractor* container should have.

Interestingly, the best performance was achieved by limiting the tool to a single thread and a single CPU, using the saved resources to spawn additional containers. In other words, 48 one-core *bulk_extractor* instances work faster than one 48-core instance. For this test case, we provided each container instance with 500 MB of file data from the GovDocs corpus (Garfinkel et al., 2009).

Table 2
SHA1 file hashing throughput (MB/s) vs. number of containers.

Containers	4	12	24	48	96	192
MB/s	345	857	985	985	948	992

Table 3
ExifTool metadata extraction throughput vs. number of containers.

Containers	4	8	32	64	96	192
MB/s	5.2	17	99	151	170	192
MB/s *per* cont.	1.3	2.1	3.1	2.4	1.8	1.0

Table 4
OpenNSFW classification throughput vs. number of containers.

Containers	4	8	12	32	64	96	192
MB/s	0.4	1.4	2.5	3.8	7.2	10.9	21.3
Files/s	0.8	2.2	3.9	7.1	13.4	20.3	38.5

Table 5
Tika text extraction vs. number of containers.

Containers	4	12	24	48	96	192
MB/s	0.5	1.1	2.4	3.5	5.8	6.7
MB/s *per* cont.	0.13	0.09	0.10	0.07	0.06	0.03

Table 6 shows that the throughput per container remains fairly stable, and exhibits linear scalability. This is not a surprise as the workload is CPU-bound and data parallel in nature.

Indexing filesystem metadata

As stated previously, SCARF utilizes *ElasticSearch* (ES) for target metadata storage as well as storage of the results produced by tasks. For this benchmark, we extracted and parsed the NTFS metadata information of a 200 GB image containing approximately 620,000 files. Each NTFS parsed record yields a JSON object of about 500 bytes.

Since ES is designed to be run as a distributed service, it is important to consider different architectures as small changes could have large performance consequences. In this benchmark, we test two architectures: a) single ES data node; and b) production-style distribution containing seven nodes of various roles.

Table 7 shows that, indeed, moving from a standalone deployment to a small cluster yields super-linear improvement as synergies among the modules and fewer performance bottlenecks improve the effectiveness of the system. We did not test larger configuration (we would need more records) but we expect larger configurations to scale out well although the per-container performance may drop slowly as with most of the other services tested.

Summary

Inspection of a simple graphical plot, show in Fig. 2, shows that additional containers improve overall throughput. We see that both ExifTool and Bulk Extractor scale very well; and, while the deep neural-network powered OpenNSFW shows a smaller rate of throughput per container, throughput does increase. On the other hand, Apache Tika throughput grows much slower - suggesting a less scalable computation. The primary takeaway is that, with the SCARF architecture, we can add as many containers as the underlying hardware allows, the primary constraint being the number of CPUs. In other words, we can *easily* apply increasing amounts of hardware to combat the ever-increasing amount of volume present in a forensic investigation.

Updated benchmarks and user interface

In addition to the full suite of benchmark on our legacy cluster, we also performed a few preliminary benchmarks on a brand new (not fully in production) cluster. We wanted to gain preliminary insight into which tasks could be run a line speed just by updating the hardware, and which ones are likely to need more careful optimization.

We chose an intermediate case by deploying 64 containers on a four-node cluster with 256RAM per node. The biggest speedup was observed for *ExifTool*, from 151 to 433 MB/s, or 2.87 times; *Tika* showed an 80% improvement in throughput, while *OpenNSFW* improved by 64%.

Table 6
Bulk extractor throughput vs. number of containers.

Containers	4	12	24	48	64	128
MB/s	3.5	17.9	22.7	54.8	59.4	151.5
MB/s *per* cont.	0.9	1.5	0.9	1.1	0.9	1.2

Table 7
ElasticSearch throughput in standalone and cluster configurations.

Containers	1	7
Records/s	1299	14,236

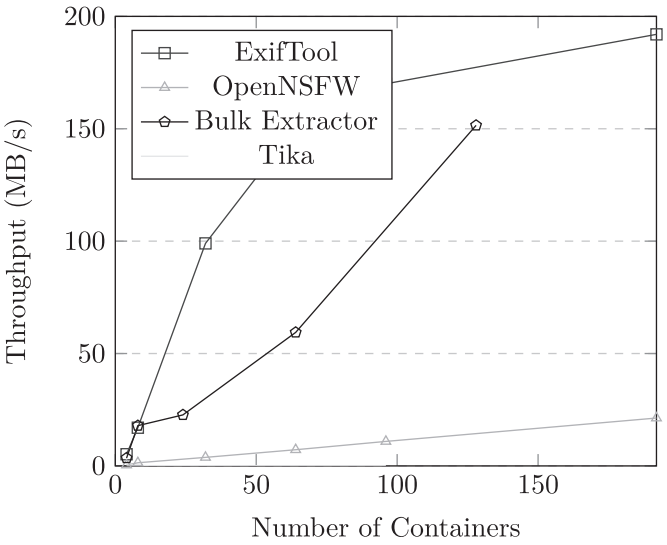

Fig. 2. Scalability of Forensic Tools using Containers.

At this early stage, the user interface is fairly simple and consists of interface with the ability to filter and drill down to the individual record. This is clearly not enough for a production tool; we believe that there is substantial room for improvement in the user interface of forensic environments, and that is a subject that deserves its own research effort and evaluation.

We have omitted a discussion on the built-in facilities of Docker and ES to monitor the performance and deal with errors and failures. These are important, but are only an existing generic administrative interface; what SCARF, and other platforms need, are both more automated means to heal the system (based on understanding the semantics of the data flows) and a better interface that speaks to the forensic task being executed in domain-specific terms.

Conclusion

In this exploratory work we have evaluated the idea using automated container deployment and orchestration platforms to achieve high performance digital forensics. Promising results show that a distributed container-based approach is a viable infrastructural foundation to address the increasing volume of data involved in digital forensic investigations. Specifically, we showed that even a modest four-node cluster can achieve very meaningful throughput improvements out of the box. For example, 150 MB/s in *bulk_extractor* processing, or 192 MB/s in *ExifTool* processing. More importantly, most of the tasks tested are naturally data parallel and show a linear, or near-linear, scale out behavior. This strongly implies that a larger cluster of 20–40 nodes could perform most current forensic processing at SSD I/O acquisition speeds (500–1000 MB/s).

Equally importantly, we showed that containers are an excellent building block for adding new processing functions to the environment at a minimal cost. In particular, we used an off-the-shelf container of a trained image classifier for adult content and incorporate it into the workflow with an hour's worth of work. Even without any optimizations, the experimental cluster was able to classify images at the rate of 3.3 million per 24 h (5.4 million on newer hardware).

The overall conclusion is that containers provide an effective and efficient platform to address both data scalability and functional extensibility for integrated digital forensic environments. We are convinced that future extensions of this work, or similar efforts, can completely change the approach to digital forensic investigations, and will provide the platform for the introduction of a new generation of intelligent processing methods.

Future work

Apart from maturing the prototype used in this work, including adding support for filesystems other than NTFS, there is a host other immediate extensions we'd like to pursue. One of the most exciting ones is experimenting with deployment on public cloud providers like AWS, Azure, or Google Cloud. Containers are seen as the replacement of many of the IaaS uses of full-stack VMs; all of these providers (and others) have already announced various levels of support for containers.

We would like to implement additional input and job modules. The input module could range from *FireWire* or *Direct Memory Access* (DMA) to input data over a wide area network. Additional functional module of interest includes various approximate matching, image analysis, and machine learning algorithms.

Containers can greatly improve the reproducibility of tool testing; indeed, even the current version of SCARF provides all the ingredients with persistent logging of tasks and results.

References

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J., 2016. Borg, omega, and kubernetes. ACM Queue 14, 70–93. http://queue.acm.org/detail.cfm?id=2898444.

Carrier, B. The Sleuthkit (tsk) and Autopsy: Open Source Digital Forensics Tools. http://sleuthkit.org.

Carrier, B., 2002. Open Source Digital Forensics Tools: the Legal Argument. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.581.6818&rep=rep1&type=pdf.

Corbet, J., 2007. Notes From a Container. https://lwn.net/Articles/256389/.

Dean, J., Ghemawat, S., Jan. 2008. MapReduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113. http://dx.doi.org/10.1145/1327452.1327492.

des Ligneris, B., 2005. Virtualization of linux based computers: the linux-vserver project. In: 19th International Symposium on High Performance Computing Systems and Applications. HPCS, pp. 340–346. http://dx.doi.org/10.1109/HPCS.2005.59.

Garfinkel, S.L., 2013. Digital media triage with bulk data analysis and bulk_extractor. J. Comput. Secur. 32, 56–72. http://dx.doi.org/10.1016/j.cose.2012.09.011.

Garfinkel, S., Farrell, P., Roussev, V., Dinolt, G., 2009. Bringing science to digital forensics with standardized forensic corpora. In: Proceedings of the Ninth Annual Digital Forensic Research Conference. DFRWS, pp. S2–S11. http://dx.doi.org/10.1016/j.diin.2009.06.016.

Harvey, P. ExifTool by Phil Harvey. http://www.sno.phy.queensu.ca/~phil/exiftool/.

Institute, N. F. Hansken, 2016. https://www.forensicinstitute.nl/products_and_services/forensic_products/hansken.aspx.

Kamp, P.-H., Watson, R.N.M., 2000. Jails: confining the omnipotent root. In: Proceceedings of the Second International System Administration and Networking Conference. SANE. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.3596&rep=rep1&type=pdf.

Linux 3.8, 2013. https://kernelnewbies.org/Linux_3.8.

LXC, 2008-17. Linux Containers. https://github.com/lxc/lxc.

Menage, P.B., 2007. Adding generic process containers to the Linux kernel. In: Proceedings of the Ottawa Linux Symposium, pp. 45–58. https://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf.

Merkel, D., 2014. Docker: lightweight linux containers for consistent development and deployment. Linux J. 2014, 239.

NVM Express Workgroup, 2016. NVM Express, Revision 1.2.1. http://www.nvmexpress.org/wp-content/uploads/NVM_Express_1_2_1_Gold_20160603.pdf.

OpenVZ Virtuozzo Containers Wiki, 2016. OpenVZ History. https://openvz.org/History.

Richard, G., Case, A., 2014. In lieu of swap: analyzing compressed ram in mac os x and linux. In: Proceedings of the 14th Annual Digital Forensic Research Conference. DFRWS. http://dx.doi.org/10.1016/j.diin.2014.05.011.

Richard, G., Roussev, V., Feb 2006. Next-generation digital forensics. Commun. ACM 49 (2), 76–80. http://dx.doi.org/10.1145/1113034.1113074.

Roussev, V., May 2011. Building open and scalable digital forensic tools. In: 2011 IEEE Sixth International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE), pp. 1–6. http://dx.doi.org/10.1109/SADFE.2011.3.

Roussev, V., Richard, G., 2004. Breaking the performance wall: the case for distributed digital forensics. In: Proceedings of the 2004 Digital Forensic Research Workshop. DFRWS. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.8692&rep=rep1&type=pdf.

Roussev, V., Wang, L., Richard, G., Marziale, L., 2009. A Cloud Computing Platform for Large-scale Forensic Computing. Springer, Berlin Heidelberg. http://dx.doi.org/10.1007/978-3-642-04155-6_15.

Roussev, V., Quates, C., Martell, R., 2013. Real-time digital forensics and triage. Digit. Investig. 10 (2), 158–167. http://dx.doi.org/10.1016/j.diin.2013.02.001.

Seagate Technology, 2016. Enterprise Capacity 3.5 HDD (Helium). http://www.seagate.com/www-content/datasheets/pdfs/ent-cap-3-5-hdd-10tb-channelDS1863-5C-1608US-en_US.pdf.

Stewart, J., 2012. Scalable forensics with TSK and Hadoop. In: Open Source Digital Forensics Conference. OSDFCon. https://www.osdfcon.org/presentations/2012/OSDF-2012-The-Sleuth-Kit-and-Apache-Hadoop-Jon-Stewart.pdf.

The Rekall Team. Rekall Memory Forensic Framework. http://www.rekall-forensic.com/.

U.S. Department of Justice, Office of the inspector general, 2015. Audit of the Federal Bureau of Investigation's Philadelphia Regional Computer Forensic Laboratory. https://oig.justice.gov/reports/2015/a1514.pdf.

U.S. Department of Justice, Office of the Inspector General, 2016. Audit of the Federal Bureau of Investigation's New Jersey Regional Computer Forensic Laboratory. https://oig.justice.gov/reports/2016/a1611.pdf.

van Beek, H., van Eijk, E., van Baar, R., Ugen, M., Bodde, J., Siemelink, A., 2015. Digital forensics as a service: game on. J. Digital Investig. 15, 20–38. http://dx.doi.org/10.1016/j.diin.2015.07.004.

Volatility Foundation, 2007–2015. Volatility Framework. https://github.com/volatilityfoundation/volatility/.