Software Provenance: Track the Reality Not the Virtual Machine

David Wilkinson
dwilk@cs.pitt.edu
University of Pittsburgh

Luís Oliveira
loliveira@pitt.edu
University of Pittsburgh

Daniel Mossé
mosse@cs.pitt.edu
University of Pittsburgh

Bruce Childers
childers@cs.pitt.edu
University of Pittsburgh

ABSTRACT

The growing use of computers and massive storage by individuals is driving interest in digital preservation. The scientific method demands accountability through digital reproducibility, adding another strong motivation for preservation. However, data alone can become obsolete if the interactivity of software required to interpret the data is lost. Virtual machines (VMs) may preserve interactivity however do so at the cost of obscuring the nature of what lies within. Occam, instead, builds VMs on-the-fly while storing and distributing well-described software packages. Thus, the system can track the exact components inside VMs without storing the machines themselves, allowing software to be repeatably built and executed. For Occam to recreate VMs, it needs to know exactly what software was used within. Through this tracking, such software can even be modified and rebuilt. Occam keeps track of all such components in manifests, allowing anybody to know exactly what is in each VM, and the origins of each component.

CCS CONCEPTS

•Applied computing → Digital libraries and archives; •Software and its engineering → Reusability; Software verification and validation; Virtual machines;

ACM Reference format:
David Wilkinson, Luís Oliveira, Daniel Mossé, and Bruce Childers. 2018. Software Provenance: Track the Reality Not the Virtual Machine. In *Proceedings of First International Workshop on Practical Reproducible Evaluation of Computer Systems, Tempe, AZ, USA, June 11, 2018 (P-RECS'18),* 6 pages.
DOI: 10.1145/3214239.3214244

1 INTRODUCTION

Software preservation is an important yet slowly evolving field of study. It was not long ago that the term terabyte was new, yet it is speculated digital data will reach 40 zettabytes by 2020 [23]. An increasingly digital generation uploading every photo and video of their daily life has certainly contributed to this boom. Therefore, a growing interest in data preservation is unsurprising.

P-RECS'18, Tempe, AZ, USA

978-1-4503-5861-3/18/06…$15.00
DOI: 10.1145/3214239.3214244

However, preserving the mere bits and bytes themselves may not be sufficient to capture the meaning of some artifact. This is already a problem with older files meant for early versions of software. For instance, when storing a Microsoft Word document, it is not enough to simply store the file. These documents do not always render as intended in newer copies of Word, and this represents a form of decay, often called *digital obsolescence*.

Much like the decay of physical artifacts, it may be difficult to know exactly when an artifact moves from interactive to unusable. In fact, it is difficult to ascertain the effect or rate of decay that might exist in future archives. Yet, it seems intuitive that any effort to mitigate decay will enable better, more sustainable digital archives.

One such solution is to progressively transform artifacts from an older format to a newer platform. In our word processing example, one could convert the document to a more generic format, such as PDF, or a more literal transcription of content and structure with a standard like DocBook XML [2, 7]. In this scheme, the archive allows the original format to obsolesce and provides interaction with the document in more modern contexts. To facilitate this, the archive choses a format to store the data and offers several formats, translated on demand, to view the content. For example, the original document is translated to a stored format, PDF or XML, and a web portal may give you the option of downloading a newer Word document, render HTML, or yield plain text.

Yet, this scheme presumes an optimism about the generality of the chosen storage format, which may itself decay or be hurt by proprietary extensions or changes, such as with PDF. Also, this is an opportunistic solution. If you discover new data, years later, in that same older format, can you still transform it? An archivist may not become aware of a loss of fidelity until after the original is unreadable. Consider data where correctness is subjective or meant to be reinterpreted in new contexts, such as scientific datasets.

In these cases, we argue that in order to best preserve data, it is best to preserve any future interaction with that data. By preserving interactive software that translates or renders content, the data itself is preserved. Interactive preservation ensures fidelity of any artifact is not lost since it promotes storing the original data, which can always be inspected. That is, transformation should not be considered the "necessary" step, and instead the community should design and promote nondestructive processes when possible.

Since software often builds upon a collaborative base of existing software, its preservation must also facilitate the practical archival and reuse of such software building blocks. In this case, the long-term preservation of any application is contingent on tracking the software that is involved in building or running programs, generating data, or visualizing data.

We illustrate the practical application of this level of software preservation by describing Occam, our free and open-source software archival system [19]. Occam not only maintains software packages but preserves the ability to build and run such software and its dependencies. Occam is a general platform to inspect and share artifacts and data, transform data, edit and deploy code, and create and deploy scientific workflows composed of multiple software artifacts. Since all data and computation is contained within the archive, any new data produced is itself preserved alongside the software used to generate it. Thus, the goal is twofold: to be attractive to researchers by being a productive tool, but also be attractive to archivists by treating preservation as a first-class entity.

Throughout this paper, virtual machines and how they are currently used by the community will be discussed. The process of generating virtual machines and the residual tracking of software within those contexts is described. In essence, we describe how Occam is able to track provenance of software artifacts and data in an archival system by direct virtue of this mechanism.

2 SOFTWARE REPRODUCIBILITY

Preserving the past is not the only concern. Technology moves quickly and requires archival practices to be proactive. To truly stave off the deterioration of information, emphasis also needs to be placed on preserving the present. Specifically, preservation and provenance need to be considered in the design of tools, especially those that are commonly used to deploy or distribute work.

When an archive preserves data integrity, the scientific community is one obvious benefactor. The scientific method, first proposed in the 11^{th} century by Ibn al-Haytham [15] and rigorously applied by Renaissance scientists, suggests science is a process that must encourage repetition, lest coincidence be mistaken for cause. In this model, accountability relies on the dedicated and accurate reporting of experimental conditions. However, science is confidently rolling toward total reliance on networks of computers in the derivation and analysis of data, in spite of it being well-known that computation and programming are highly prone to error [8]. Considering errors could be discovered well after publication, it is important the readability of any related data is preserved. Digital obsolescence is, then, an antagonist to scientific repeatability.

There have been many recent efforts to determine the scope of digital research and provide insight on the accountability or lack thereof. In the UK, researchers surveyed said that modern research is impossible without software [14]. Yet, in a reproducibility study within the field of Computer Science (CS), which is obviously dominated by software-backed research, shows that only 32% of recent CS research is reproducible [4]. That is, digital results of typical CS studies cannot be replicated. Other fields have similarly raised this alarm: 90% of scientists of varied disciplines surveyed by Nature opined there is a slight or significant reproducibility crisis [1].

With respect to scientific repeatability, one strategy is to distribute a virtual machine (VM) image for the artifact. First, the software dependencies (libraries and even the operating system itself) are contained within the VM and configured, mitigating a particularly chaotic problem of software distribution. Also, distributing the artifact is simple since a virtual machine can be packaged as a single file using the Open Virtualization Format (OVF) [10] specification.

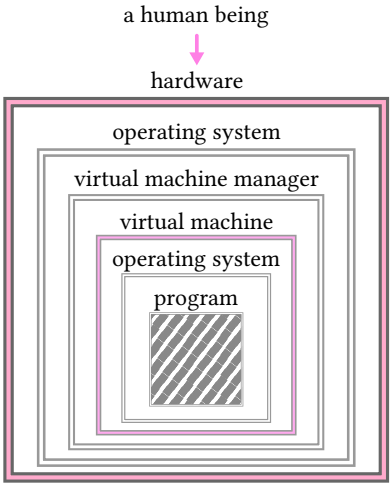

Figure 1: The virtualization stack required to run a piece of software. The artifact exists within the virtual machine through several layers. The contents of the machine are arbitrary, difficult to access, and typically not well-described.

These VM images can run on a variety of systems using VirtualBox [22] or VMWare [24], and distributed on Zenodo [25], Dropbox [9], the Open Science Framework [21], or similar services.

Since the goal of OVF is to provide compatibility among a set of virtualization tools, the format can describe VM characteristics quite well. However, it does not help describe the application software being executed. Consider the typical layout of a virtualized system shown in Figure 1. Here, the virtual machine manager is a tool such as VirtualBox. The hardware underneath the VM is provided by the individual interested in the artifact. The contents within the virtual machine may be vastly different from one virtual machine to another. The VM abstraction obscures valuable information about the historic interactions between the software within the VM and the software used to build it.

Consider, within a scientific domain, analysis software with a certain level of accuracy. In this case, if there was an error caused by an outside tool (e.g., a different library version) that made the analysis tool less accurate, this error would be obscured since the VM only gives us a view of the artifact as a whole. Such errors can even occur in widely used code, such as a standard library [13], which people tend to assume is correct. However, there is no requirement that metadata is provided about its compilation or even the packaging of the compiler within the VM. If this information were known, scientific work could be tracked and reevaluated when relevant compiler bugs surface.

While a virtual machine abstraction helps the distribution of software, it hinders efforts to understand that software. Our thesis is that this is an unreasonable and unnecessary trade-off. Why distribute a VM with a description of its contents if you could build a VM from that description? Since the product is still a VM, this can only offer greater flexibility at no cost to the quality of preservation or ease. That is, it is much more flexible to track software used in building and running artifacts and creating (and

recreating) virtual machines only when needed. This will ensure the long-term usability of software and data in digital archives.

3 FROM PROVENANCE TO VM

Occam takes the approach that if repeatably executing artifacts is the goal, then virtual machines are the technology, not the result. As discussed, OVF and similar formats around containerization are not designed around artifact preservation and obscure details about the software executed within. However, software artifacts should be distributed along with as much information as necessary to recreate them. For instance, with properly described software, one could generate the necessary virtual machine manifest using the recorded provenance in a repeatable manner. Distributing the entire virtual machine image is then unnecessary as this provenance is its equivalent. Instead, the problem changes to a software storage and distribution problem similar to the spaces occupied by distribution protocols and package managers.

To this end, Occam preserves software as an independent artifact, which is defined by a set of metadata used to generate a virtual machine. The main identifier for an object is a multihash in the style of IPFS [3] computed with the name and type of the artifact and its initial git commit. While a thorough discussion of these identifiers is interesting, it is beyond the scope of this paper. Briefly, this style of hash is useful for federated discovery. When the manifest is generated, it will contain the hashes of each used artifact, which can be used to lookup and verify the original artifact.

There are informative, human-readable fields such as name, description, tags, and authors, which are each useful for discovering existing artifacts and attribution. The remainder of the fields are used to build virtual machine manifests. These include values representing the expected environment and architecture, which help the manifest generator decide which backend technology can be used. There are fields for listing dependencies that link the artifact to a variety of other software artifacts: software libraries, tools, or non-executable entities such as datasets or configurations.

In Listing 1, the abridged metadata that describes gcc, a common C compiler, is shown. The `environment` and `architecture` fields are used to generate a VM capable of providing that system environment. In this case, a 64-bit x86 machine with a Linux kernel. The metadata is then divided into sections `build` and `run` for describing how the artifact is built and executed, respectively. Using these sections, a separate VM is created for building and running an artifact. This makes the system very flexible since a VM is generated with only the exact dependencies required. The dependencies required to build the object can be omitted from the virtual machine generated to run the object, and vice versa, reducing the burden of deploying either. Furthermore, due to storing the short manifest, described in the next section, that represents the machine, the virtual machine to build the object can be generated at any point. Meaning, it may be destroyed after its use.

When a person chooses to build or run an object, its metadata is given to Occam's Manifest Manager, which takes an object and produces a virtual machine manifest. To do this, it analyzes the metadata for its environment and architecture and determines the other artifacts within the archive which are required to provide this system. In the case of gcc, the system it requires is very similar to the native system. However, we can also support now-obsolete systems, for instance a "dos" environment in reference to the operating system of many 1980s PCs. In these cases, the Manifest Manager would select a DOS emulator and add this emulator to the VM to bridge the required "dos" environment to the, let's presume, native "linux" environment of the virtual machine manager.

```
{
 "type": "compiler",
 "name": "gcc",
 "environment":  "linux",
 "architecture": "x86-64",
 "build": {
  "dependencies": [ {
   "id": "QmUpBXwVhGpWzaKToh6jxQJQxGbJvZQxFWJ16nHK5WXsWd",
   "type": "language",
   "name": "bash",
   "version": "4.x"
  } ],
  "command": ["/bin/bash", "build.sh"]
 },
 "run": {
  "command": "usr/bin/gcc"
 }
}
```

Listing 1: This object, gcc, is a compiler that might be included in a virtual machine to build another piece of software. This metadata describes its platform requirements, build dependencies, and how to invoke a build or run.

Since Occam does not treat VMs as the product, it is free to determine how to run an artifact at the precise time it is needed. The choice of emulator is not hard-coded. Any piece of software can announce that it provides "dos" and be considered in the manifest. In the same manner that a DOS emulator helps us provide the particular "dos"/"x86" system, in the future a similar tool can provide a "linux"/"x86-64" system for current artifacts. That tool, like the emulator, will be stored within the system as a software artifact itself and preserved like every other. In this scheme, there is no reliance on supporting virtual machine managers, such as VirtualBox, in order to maintain long-term software preservation.

The idea of having software descriptions as the core mechanic to preserving software is applied to as much of the system design as possible. In fact, when a virtual machine manifest is generated, the result itself is a software artifact. Metadata describing any archived software is, in a sense, an incomplete manifest which may be instantiated as one or many different virtual machines depending on the available technology or hardware. In comparison, VM manifests make use of the same metadata fields as typical artifacts and are considered complete as their target environment and architecture match the host machine. Furthermore, the manifests have details about every piece of software or data residing within the VM. When Occam executes a manifest, it uses the description to create and deploy such a VM using the available technology, such as Docker or VirtualBox. Future technologies (the next Docker or VirtualBox) are supported simply by writing a plugin that reads a manifest and executes the appropriate commands or configurations required.

For example, Docker support involves a software artifact that would represent the base container. Listing 2 shows a piece of a manifest to build gcc. Here, the base Docker container is the main

software object that runs. This object provides the "linux"/"x86-64" system environment and is responsible for running the objects dictated in the manifest. In this case, it will mount bash and run the provided shell script to initiate the build.

More technically, this Docker base object simply has a boiled-down python environment with an initialization script. This script is responsible for ultimately running the software present in the container and runs within the virtual machine. It will simply prepare and execute the intended software artifact. In our VirtualBox environment, this docker environment is ignored and, instead, has its own initialization script, yet a similar process occurs. This flexibility allows for nesting a variety of environments within a single VM to allow obscure software environments the best chance of being usable on any particular system.

Referring back to the Word document example, we could have WordStar as an artifact. When the system is told to view a document, it looks at providing a DOS environment for Wordstar: DOSBox. The system then recurses, asking for the more complete manifest for running DOSBox. In a sense, the manifest provides a path from the host hardware to the requested software through various artifacts in the archive. It would run the Docker initialization script, which would yield to the DOSBox initialization script, which would run the requested software, which opens the requested file. A similar process would happen if you wanted to open that document in Microsoft Word or use a different, perhaps more accurate, emulator. When you change these choices, you would get, on the fly, a different manifest. This flexibility allows us to stave off digital obsolescence by continuously being able to reinterpret the required environments to execute software artifacts.

4 BUILDING THE PROVENANCE ARCHIVE

The manifests described are generic enough to be useful to both Docker and VirtualBox, but their intended purpose goes beyond mere execution. Wrapping artifacts in Occam ensures every invocation is traceable. Indeed, the manifest contains links back to the exact versions of artifacts used within. Furthermore, there is a link, when necessary, to the manifest of the VM that built each artifact. One can investigate the origin of each and every artifact used, not only in the intended VM, but within any VM used to prepare dependent software.

For instance, when inspecting a container and noticing it used a library called "libghost", one will find the identifier of the manifest that built that library under a `build` key. The manifest, generated in the past when the object was built, will have a record of "gcc", a compiler, and point to the exact manifest that built that particular compiler, perhaps one like Listing 2 from the previous section, which may then point you to Listing 1. Of course, all components can be inspected: the manifest for the built compiler, each library required by that build process, and so on.

Each object in Occam maintains a history of changes by virtue of being stored with `git`, a common version control tool [12]. The `revision` refers to a point in time and is represented as a digest and tagged as provenance metadata in the manifest for each object. If a manifest is executed a second time, revision tags ensure it will use the same versions of software as the first. Some artifacts have data less suited to `git`, which is designed for text such as source code.

```
{
 "type": "task",
 "name": "Task to build gcc on docker",
 "environment":  "docker",
 "architecture": "x86-64",
 "running": [{
   "id": "QmPQogaMrcNBP6DKDkNsKjkgAw5Jz7Emk53WbAFDAc8xdZ",
   "type": "docker-environment",
   "name": "occam",
   "revision": "5dqyHCUNTU3QEzW2zB6NTz7fRr4w3A",
   "running": [{
     "id": "QmUpBXwVhGpWzaKToh6jxQJQxGbJvZQxFWJ16nHK5WXsWd",
     "type": "language",
     "name": "bash",
     "revision": "5dr2FNjudPfa6cmZ8AWQUWSPZi4mj6",
     "build": {
       "id": "Qmb7gGofQcXpDFx53rUrS36joNMwouAbtZu2yYhPgKr1aj",
       "revision": "5dqwk4YjJ2FLE4sTUjByEDGszVFS2Y"
     },
     "paths": {
       "mount": "/occam/QmUpBXwVhGpWzaKTo...GbJvZQxFWJ16nHK5WXsWd"
     }
   }, {
     "id": "QmTE53bYFjoEgj4M8YQo9CKuqCKTk4xuwUz6U5jP58jKZE",
     "type": "compiler",
     "name": "gcc",
     "revision": "5du3YqiR1diUtrHFGVCoojNXMoMXnN",
     "run": {
       "command": ["/bin/bash", "build.sh"]
     },
     "paths": {
       "mount": "/occam/QmTE53bYFjoEgj4M...CKTk4xuwUz6U5jP58jKZE"
     }
   }]
 }]
}
```

Listing 2: An abridged and paraphrased task manifest for building gcc. In reality, there are typically dozens of dependencies. One such dependency may even be an earlier version of gcc, which would link to its own task manifest.

Large files are, instead, tagged via a mathematical hash of their content, which is recorded in the manifest. This information can indicate two independent experiments are using the same dataset.

This tracking of provenance seems severe. However, it comes at very little cost. A manifest is a simple JSON formatted file (a few kilobytes) and can be stored and distributed alongside the related artifact. The built binaries, in comparison, are vastly larger, often in megabytes or even gigabytes. In the case of gcc, which is one of our largest archived applications, the source code distributable is 101 MiB, the built binaries are 277 MiB, but the task manifest is a mere 64 KiB. When distributing an artifact, it therefore costs very little to also distribute the accompanying tree of manifests.

This tracking may also be seen as superfluous. However, the knowledge of which compiler was used can be instrumental in understanding software errors. It may be true that compiler options change the behavior of software in an unintended manner. Often, this information is lost when distributing software through packaging binaries in archive formats, such as tar or zip. In Occam, we can directly inspect that compiler choice and even, through the system, build dependencies with different compiler versions to inspect the difference. Similarly, older experiments or artifacts can be built using newer compilers, perhaps automatically or at random, and their output compared against a previous invocation.

In these cases, any inconsistencies can be reported and reviewed in case of either a new flaw or the discovery of an existing flaw that was recently fixed.

To facilitate this tracking, Occam must maintain its own copies of software. This is arguably an ambitious task. However, the software archive has been built around existing package managers, since they are built around similar, albeit somewhat shallower, metadata descriptions. For our purposes, we used the Arch Linux package repository as a starting point. In the Arch User Repository, each software package has some descriptive metadata and a build script. These are manually translated to Occam because Arch Linux allows for cyclic dependencies and presumes a more complete system. However, when the archive becomes similarly complete, it is possible to use Arch packages to update existing objects within the archive automatically. Currently, in spite of this manual effort, Occam houses hundreds of software artifacts which were all built within Occam, and built off of each other such that every piece of software is being tracked, preserved, and versioned.

Other package repositories were used to load the archive with libraries from other important communities. Occam has a plugin system to facilitate the conversion of packages from these external sources. For instance, we easily create Occam objects for any library within PyPI, the Python community's repository. This was successfully used to archive a runnable copy of Jupyter Notebook, and thus useful to preserve any notebook by simply uploading the `.ipynb` file as an artifact. Each Python library is tracked along with provenance information, such as the exact URL from PyPI where the package was initially imported. If such a resource, commonly a tar file in the case of a Python library distribution, is pulled from an external source, a hash of its content is tracked in any manifest. Precautions are taken if the external resource disappears or changes. The archive keeps a copy of that external resource and is distributed when it is needed, even if destroyed at its origin, and verified by the content hash.

Scientific accountability is preserved by having generated data (e.g. tables or graphs) point to the software that generated it. If we gathered data from an analysis tool, that data would point back to the tool, its manifest, and thus every piece of software involved. In our Word document example, if we used software to produce a Word document, that document would, similarly, link back to the exact version of Word that produced it. In this world, when given a copy of a document, that link would carry enough information to run the word processor in order to open the document as intended. Due to the flexibility of the VM generation, the task manifest of the author may be reused. In the case that the reader has, perhaps, a future machine that didn't exist in the author's time, a new manifest can be created. If a different application is desired to render the file, a new manifest could be generated. Of course, this hypothetical document could be a graph, an obscure dataset, an older PDF, or even represent an HTML document or webpage where "Word" is analogous to an older version of Firefox or Chrome.

5 THE WELL-DESCRIBED WORLD

By accurately and completely describing software, including the exact tracking of its environment, we can offer long-term preservation. Yet, there is great potential to innovate. For instance, attribution of derived work can very accurately describe each and every software and data contribution. These citations can be automatically generated by looking at the virtual machine manifest. This manifest can be tagged on the output of workflows including graphs, tables, and even papers. This gives us a vision of a much more interactive publishing platform where data links back effortlessly to the software configuration that produced it.

Occam borrows much of its philosophy from package managers, such as npm [18] and aptitude [6]. These systems, due to the responsibility of pulling in software for critical systems, have provenance tracking mechanisms to verify the origin and correctness of said software. This leads us to the interesting prospect that software archival reduces to the same problem space as package management, a position supported by similar projects such as Spack [11]. Therefore, Occam, and similar archives, are well-engineered to become the package managers for not only the scientific space, but for the general public. This is strengthened by any effort to federate the system such that software artifacts are widely available and the archives themselves are decentralized.

In the end, separating the VM from the software artifact allows flexibility in choosing the virtualization platform. One future benefit from this design is the VM image can be changed or optimized. Typically, a VM includes a fixed Linux distribution (e.g., Ubuntu) in the image. However, with Occam, and similar manifest-based systems such as Umbrella [17] or Collective Knowledge [5], at any point this choice can be explored and new platforms could be developed and measured. For instance, a specialized unikernel approach similar to efforts such as MirageOS [16], where you reduce the operating system environment to only what is needed to run the application, is impractical in a generalized context. However, when part of an automated scheme, it now seems reasonable and would likely reduce the footprint and requirements of the virtual machine.

Occam is currently focusing on decentralized distribution and collaboration. Given that software is preserved, how best to allow its discovery? If you import or dust-off data at one institution, how do you discover the software required to understand it that may have been introduced at another? To that end, our design looks toward incorporating federated systems ideas and technologies, such as IPFS [3] and dat [20]. Consider a network of software, data, and the knowledge of their relationship within a widespread yet cooperative network. Already, the system can import data and determine, without any local knowledge, a list of compatible software and information about how to run that software.

It is here that the community can innovate. Not just to promote digital preservation, but to envision completely new systems that inherently preserve their own ability to compute, and then recompute. Systems that break out of the virtual. Systems that track provenance, and by doing so allow anyone to tear something apart, inspect how it works, how it was constructed, and put it back together again. Systems that can be confidently used by librarians, archivists, scientists, and, perhaps, everybody else.

Acknowledgements: The material in this document is based in part upon work supported by the National Science Foundation (NSF) under grant numbers ACI-1535232 and CNS-1305220. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

REFERENCES

[1] M. Baker. 2016. 1,500 scientists lift the lid on reproducibility. *Nature* 533 (May 2016), 452–454. DOI: http://dx.doi.org/10.1038/533452a

[2] Ian Barnes. 2006. Preservation of word processing documents. http://apsr.anu.edu.au/publications/word_processing_preservation.pdf, (14 7 2006). http://apsr.anu.edu.au/publications/word_processing_preservation.pdf

[3] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *CoRR* abs/1407.3561 (2014). arXiv:1407.3561 http://arxiv.org/abs/1407.3561

[4] Christian Collberg and Todd A. Proebsting. 2016. Repeatability in Computer Systems Research. *Commun. ACM* 59, 3 (Feb. 2016), 62–69. DOI: http://dx.doi.org/10.1145/2812803

[5] Collective Knowledge 2016. Collective Knowledge. https://github.com/ctuning/ck. (2016). [Online; accessed 29-Aug-2016].

[6] Debian. 2018. Aptitude. https://wiki.debian.org/Aptitude. (2018). [Online; accessed 22-Mar-2018].

[7] DocBook 2018. DocBook Schemas. https://docbook.org/schemas/. (2018). [Online; accessed 9-Apr-2018].

[8] David L Donoho, Arian Maleki, Inam Ur Rahman, Morteza Shahram, and Victoria Stodden. 2009. Reproducible research in computational harmonic analysis. *Computing in Science & Engineering* 11, 1 (2009), 8–18.

[9] Dropbox 2018. Dropbox. https://dropbox.com/. (2018). [Online; accessed 22-Mar-2018].

[10] Distributed Management Task Force. 2015. *Open Virtualization Format Specification 2.1.1.* Technical Report. https://www.dmtf.org/sites/default/files/standards/documents/DSP0243_2.1.1.pdf

[11] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. DOI: http://dx.doi.org/10.1145/2807591.2807623

[12] Git 2014. Git Protocols. http://git-scm.com/book/en/v2/Git-on-the-Server-The-Protocols. (2014). [Online; accessed 05-Nov-2014].

[13] Tristan Glatard, Lindsay B. Lewis, Rafael Ferreira da Silva, Reza Adalat, Natacha Beck, Claude Lepage, Pierre Rioux, Marc-Etienne Rousseau, Tarek Sherif, Ewa Deelman, Najmeh Khalili-Mahani, and Alan C. Evans. 2015. Reproducibility of neuroimaging analyses across operating systems. *Frontiers in Neuroinformatics* 9 (2015), 12. DOI: http://dx.doi.org/10.3389/fninf.2015.00012

[14] Simon Hettrick, Mario Antonioletti, Les Carr, Neil Chue Hong, Stephen Crouch, David De Roure, Iain Emsley, Carole Goble, Alexander Hay, Devasena Inupakutika, Mike Jackson, Aleksandra Nenadic, Tim Parkinson, Mark I Parsons, Aleksandra Pawlik, Giacomo Peru, Arno Proeme, John Robinson, and Shoaib Sufi. 2014. UK Research Software Survey 2014. (Dec. 2014). DOI: http://dx.doi.org/10.5281/zenodo.14809

[15] Hasan Ibn al-Husain Ibn-al Haitam and Sabra A. I. 1989. *The optics. on direct vision.* The Warburg Institute, Univ. of London.

[16] Anil Madhavapeddy and David J. Scott. 2013. Unikernels: Rise of the Virtual Library Operating System. *Queue* 11, 11, Article 30 (Dec. 2013), 15 pages. DOI: http://dx.doi.org/10.1145/2557963.2566628

[17] Haiyan Meng and Douglas Thain. 2015. Umbrella: A Portable Environment Creator for Reproducible Computing on Clusters, Clouds, and Grids. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing (VTDC '15)*. ACM, New York, NY, USA, 23–30. DOI: http://dx.doi.org/10.1145/2755979.2755982

[18] npm. 2018. npm. https://docs.npmjs.com/. (2018). [Online; accessed 22-Mar-2018].

[19] Occam 2018. Occam – Open Curation for Computational Artifact Management. https://occam.cs.pitt.edu/. (2018). [Online; accessed 9-Apr-2018].

[20] Maxwell Ogden, Karissa McKelvey, Mathias Buus Madsen, and Code for Science. 2017. Dat - Distributed Dataset Synchronization and Versioning. https://github.com/datproject/docs/blob/master/papers/dat-paper.pdf. (May 2017).

[21] Open Science Framework 2016. OSF — Home. https://osf.io/. (2016). [Online; accessed 29-Aug-2016].

[22] Oracle. 2015. VirtualBox. http://www.virtualbox.org. (2015). [Online; accessed 10-Jan-2015].

[23] J. Gantz D. Reinsel. 2012. The digital universe in 2020: Big data bigger digital shadows and biggest growth in the far east. (2012). https://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf

[24] VMWare 2012. VMware Player. http://www.vmware.com/products/player/. (2012). [Online; accessed 11-Sep-2012].

[25] Zenodo 2016. Zenodo. https://zenodo.org/. (2016). [Online; accessed 29-Aug-2016].