The Data Publish Tool: A Provenance Client

Greg Janée, James Frew, and Peter Slaughter

Earth Research Institute
University of California, Santa Barbara
Santa Barbara, CA 93106-3060
{gjanee,frew,peter}@eri.ucsb.edu

Abstract. The publication of a data product is the key point at which its quality and fitness are asserted. It is also the point at which several data management tasks are commonly undertaken, such as assigning persistent identifiers, assembling citation and other metadata, and moving files to public-facing delivery systems.
We describe a "publish" tool that provides hooks for implementing these publication actions. The tool is driven by a combination of: the data product's provenance, as recorded by the ES3 system; a set of customizable publication rules; and manual review, annotation, and "endorsement" by the publishing scientist. The record of the tool's actions becomes part of the data product's permanent metadata.
We give an overview of the design and operation of the tool and describe some of the challenges that arose in its development, from the importance of context in interpreting provenance to increased demands on automatic provenance capture systems.

Keywords: provenance, publishing, curation

1 Introduction

Scientists are increasingly being called upon to publish the data supporting their conclusions. And as the science of data informatics matures, publishing such data requires addressing an increasing array of curation-related concerns: moving the data to preservation-supporting, redundant storage; assigning a persistent identifier to the data to support long-term identification; creating citation metadata to support the data's referential ability in the larger scientific record; and, last but not least, providing explicit statements of the quality, processing steps, and provenance of the data to allow future users, who may be completely unfamiliar with the original creator and context, to understand the data and its appropriate uses and limitations.

Yet these added burdens and formalism can be at odds with how data is produced in the first place. Scientists may use relatively unstructured environments (the Unix command line environment, for example, in concert with *ad hoc* data processing tools) in which hypotheses can be quickly posed and tested, experiments assembled and run, and algorithms refined and calibrated. Scientists are as likely to use one-off scripts, legacy programs, and volatile collections of

data and parametric assumptions as they are to frame their investigations using formally-specified workflows.

How can we bridge these two worlds, one in which algorithms are developed and refined on the one hand, and one in which algorithms are preserved, documented, and permanently associated with the data they generated on the other? We believe that a "publish" tool, invoked at the time of publication and even providing the means of publication, can act as a natural bridge in this situation. Such a tool can provide the basic platform for handling the aforementioned data, metadata, and provenance management tasks.

Regarding provenance specifically, our basic assumption is that a large part of what determines a data product's fitness for publication is the process by which that data product was produced, and the components which were combined or transformed in order to produce it. A data product produced by software with known significant bugs, or from source datasets with known errors, may not be suitable for publication, regardless of how well formatted, documented, and presented the data product is. Or, even if the software and source data used to produce a data product are demonstrably acceptable, they may not be suitably curated, and this lack of a guaranteed level of future availability to users of the data product may be sufficient to disqualify it for publication. Thus provenance plays a central role in the decision to publish, and in the assertion of suitability implied by the very act of publishing.

In previous work [5] we hypothesized a system that, given the request to publish a particular data product, traverses that product's provenance and applies rule-based tests to each of its computational antecedents to determine whether the data's provenance is robust enough to justify its publication. In this paper we describe a first version of such a tool. Our "publish" tool utilizes provenance automatically gathered by the ES3 provenance system, and allows the user to examine, annotate, and ultimately endorse the provenance for a data product to be published.

In sections 4 and 5 we describe the tool's basic operation and walk through an example invocation, and in section 6 we discuss some of the issues that arose during the tool's development. But first, we give a brief background of the ES3 system and the role of the publish tool in the context of ES3.

2 ES3

ES3 [3, 4] is a software system that can automatically and transparently capture, manage, and reconstruct the provenance of arbitrary, unmodified computational sequences. Specifically, ES3 captures provenance from running processes, as opposed to extracting or inferring it from static specifications such as wrappers or workflows. ES3 provenance management can thus be added to existing computations without modifying or re-specifying them.

ES3 models provenance in terms of processes and their input and output files. We use "process" in the operating system sense; i.e., a specific execution of a program. Each program execution or file access yields new provenance events.

Relationships between files and processes are inferred from whether files or interprocess connections are accessed for reading or writing. System call tracing allows ES3 to function completely independently of the scientist's choice of programming tools or execution environments.

An ES3 provenance document is the directed graph of files and processes resulting from a specific invocation event (e.g., a "job"). Nested processes (processes that spawn other processes) and interprocess communication (e.g., pipes) are correctly represented. ES3 supports arbitrary forward (descendant) and/or reverse (ancestor) provenance retrieval, starting at any specified file or process.

ES3 is implemented as a provenance-gathering client and a provenance-managing server. The client, which runs in the same environment as the processes whose provenance is being tracked, comprises a *logger* process, that intercepts raw system call traces and writes the provenance-relevant events to log files; and a *transmitter* process, that asynchronously scans the log files, assembles the provenance events into a time-ordered stream, assigns UUIDs to each file and process being tracked, and submits a raw provenance report to the ES3 server.

The ES3 server is a PostgreSQL[1] database with a web service layer that supports insertion of file and provenance metadata, and retrieval of provenance graphs. File metadata allows ES3 to track the one-to-many correspondence between external file identifiers (e.g., pathnames) and references to those files by UUIDs in provenance reports. Provenance queries cause the ES3 server to assemble a provenance graph (by linking UUIDs) starting at a specified process or file and proceeding in either the ancestor or descendant direction.

3 Publication in the context of ES3

Although ES3's provenance capture is entirely automated, we find there are three provenance-related actions, possibly involving human intervention, that are usefully performed at data publication time:

- **Culling extraneous provenance**: The publication of a data product provides a natural opportunity to cull extraneous provenance. ES3 is capable of recording all of a user's actions, including repeated computations of the same data product. The publish tool can provide ES3 with a trigger to cull provenance graphs not worth saving, and to flag the ones that are worthy of preservation.
- **Checking completeness**: The publication of a data product provides an opportunity to check that the provenance has in fact been gathered. ES3's tracing can be enabled at will, and it may not have been enabled during certain stages of the product's creation. For example, it may be revealed that the provenance or documentation of some input data files is missing, or that the provenance linking source code files to executable binaries is missing, or that uncommitted (with respect to a revision control system) source code files were used in the product's creation.

[1] `http://www.postgresql.org/`

– **Annotating the provenance graph**: ES3 captures provenance at the operating system level (process execution and file I/O) and thus transformations and files are identified in operating system terms: pathnames, checksums, and last-modified timestamps. But while identifying a file by pathname and checksum is unambiguous and effective for ES3's purposes in linking nodes into a provenance graph, it is of limited value to the human reader. Knowing that a file is named `3em.veg3rms.pic` and has checksum `3b7d80a6432122dc0f7f7057f57ee949` gives little clue to its real identity and purpose. The publish tool provides an opportunity, through annotations that ultimately get stored back in the provenance graph, to map such ES3 identifiers to more descriptive terms. In the case of an input data file, an annotation might record the source of the file (e.g., "SeaWiFS chlorophyll product version 5.2 obtained from NASA GSFC on…"). In the case of a transformation, the annotation can describe its nature, sometimes quite succinctly (e.g., "MATLAB R2007").

4 Publish tool design and operation

Our publish tool is invoked from the Unix command line and starts with a candidate data product to be published, which is identified by pathname. A provenance trace for the file (as it currently exists) must be present in ES3 to proceed.

The publish tool's ultimate goal is to "endorse" the data product's node in the provenance graph and, by exension, the data product itself. Endorsement here can be seen as an affirmation of the product's fitness for publication (see section 6.1 for further discussion of the meaning of endorsement). Endorsement is also transitive in that a provenance graph node is only provisionally endorsed unless all antecedent nodes are endorsed as well. Thus the publish tool performs a depth-first, post-order traversal of the candidate data product's backward-looking provenance graph, addressing the endorsement of each node in turn. At each step, the user is asked to endorse the node and optionally provide an annotation. Annotations are unconstrained, but the intention is that they be used to document the source of input data files and the nature of transformations. Should the user not be prepared to endorse the node at that time, the node can be skipped; all subsequent nodes will only be provisionally endorsed, but the publish tool caches intermediate work, so a repeated run visits only the unendorsed nodes. The user can also ask that a node be ignored, presumably because it is of no consequence for the long-term understanding of the data product.

The publish tool as described thus far would be onerous to run, but fortunately it features a rule system that partially, if not completely, automates node endorsement. The rules presently supported include two built-in rules:

– **Transitivity**: If a transformation is endorsed, and if all its file inputs are endorsed, then all of the transformation's file outputs are endorsed. The

basis for this rule is simple determinism: an acceptable function applied to acceptable inputs must, by definition, create acceptable outputs. This rule has the effect of removing temporary and other intermediate files from explicit consideration.
- **Presence in revision control systems**: If a file is found to be registered in a revision control system (e.g., CVS[2] or Mercurial[3]) repository, and if the file's contents at the time it was accessed match the contents of some revision of the file in the repository, then the file is automatically endorsed. The annotation supplied by the publish tool in this case is the repository location and the revision number corresponding to the file's content. (Determining revision numbers is discussed in more detail in section 6.3.) If the file is found in a repository but its contents do *not* match any revision (i.e., the provenance referenced an "uncommitted" file), then the publish tool alerts the user to this fact. The user is given the option of overriding the warning with a manual endorsement, but this should be seen as a mistake in most cases, as the use of uncommitted code defeats any linkage of provenance to revision control history.

And one category of user-defined rules:

- **Glob pattern match**: If the file named by a node (in the case of a transformation, this is the transformation's executable file) matches a user-defined extended glob pattern[4], then the node is automatically endorsed and given an annotation as specified by the rule. Rules are created by the user, and to ensure that the invocation of a rule matches the user's intention, the publish tool caches a list of all files matching the glob pattern, along with their checksums, at the time of rule creation. At the time a rule is invoked, the file in question is checked against the cache; if it matches, the rule proceeds, but if the check fails (either because the file is not in the list, or the file's current checksum does not match that stored by the rule), the user is alerted to this fact and given an opportunity to update the rule, to manually endorse the node, or to skip the node and address the problem in some other way. In this way the publish tool helps detect unexpected changes to files.

The use of rules eliminates much if not all of the manual process of endorsing nodes, as will be demonstrated next.

5 Example invocation

To look at a concrete example, let us consider an invocation of the publish tool to publish a MODSCAG snow coverage product [7] granule that was produced from a MODIS surface reflectance product granule. In this example we assume ES3

[2] `http://savannah.nongnu.org/projects/cvs`
[3] `http://mercurial.selenic.com/`
[4] `http://ant.apache.org/manual/dirtasks.html#patterns`

has been used to trace the compilation of the MODSCAG source code as well. The resulting provenance graph [2] is too large to display in this space, containing 434 nodes. Fortunately, the publish tool's rule system cuts this number down quickly:

- 392 node are intermediate file nodes, and hence are automatically endorsed by transitivity.
- 9 nodes represent MODSCAG source files. Assuming these are registered in a revision control system, their endorsement is automatic.
- 30 nodes represent transformations and transformation-related files (e.g., the gcc compiler and the IDL interpreter) whose endorsement can be automated using 5 rules covering various system directories (e.g., `/bin/**/*`).
- 1 node represents the input MODIS granule. Either this node must be manually endorsed, or a rule endorsing all such files in the same directory can be specified.
- 1 node represents the output snow coverage, and is automatically endorsed by transitivity.
- Finally, 1 node represents a shell script that automates the running of the MODSCAG program, and contains various input parameters. At the user's discretion, such a node may be marked as ignorable, either manually or by a rule.

Thus, running the publish tool for the first time, assuming no existing rules, the user is required to examine 7 out of the 434 nodes. But if we assume that rules covering system directories are already in effect (i.e., have been put in place by some previous run of the publish tool; see section 6.1 for a discussion of rule storage locations), the number of nodes that must be examined drops to 2. And if the user publishes another MODSCAG granule similarly computed, and if rules have been appropriately defined, the number of nodes requiring manual evaluation drops to zero, i.e., publication is entirely automatic.

6 Issues and discussion

The development of the publish tool brought up a number of issues and challenges, which we discuss here.

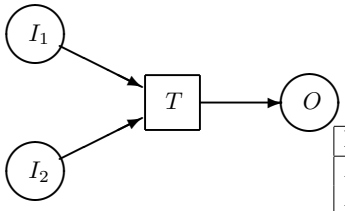
6.1 The role of context in endorsement

As described in section 4, the publish tool's basic *modus operandi* is to endorse provenance graph nodes, starting from earliest antecedents and ending with the data product proposed for publication. There is, however, a subtle difference in what endorsement means between antecedents and the final data product.

An endorsement of a data product is a statement of its quality and fitness for publication, but an endorsement of an antecedent is a statement of the fitness of the antecedent only in so far as it contributes to the data product. Put another

way, the endorsement has meaning only in the context of the data product that initiated the publication process. That this is the case is easily seen if we consider, for example, a data product that uses only a subset of an antecedent, say, only a certain cloud-free portion of a remote sensing image. The endorsement of the antecedent in this case is a statement that the antecedent is fit for the data product's purposes, and in particular, that the portion used is cloud-free. But such an endorsement does not imply that any other portions of the scene are cloud-free, nor does it have any applicability to other uses of the antecedent in which cloudiness is desirable or simply not a factor. Thus endorsement is a combined property of both the antecedent and the published data product, and thus, in publishing a data product, every antecedent of the product must be examined. (Shielding users from this consequence was the very reason for the development of endorsement rules.)

This distinction between antecedent endorsement and publication endorsement is architecturally reflected by binding all endorsements supporting a data product's publication to the product's provenance node in ES3 as shown in Figure 1 below. The complete publication annotation includes endorsements, comments, rules invoked, and information about who did the publishing.



Node	Endorsement
I_1	by rule...
I_2	obtained from...
T	IDL 7.1
O	by transitivity

Figure 1. Example placement and content of a publication annotation in a provenance graph.

Of course, a previously published data product may serve as an antecedent to another data product. The fact that an antecedent was itself previously published (and therefore has its own publish tool annotation in ES3) *may* be used as a basis for endorsing it in the context of publishing another data product, and in this way publication-level endorsements can be transitive. However, this transitivity is not automatic for the reasons listed above: the context of publication is different and a re-evaluation is required. We are still developing rule mechanisms that would allow previously published data products to be automatically endorsed under certain conditions (e.g., if published by the same user). Architecturally, the endorsement of an antecedent that happens to be previously published is treated no differently than any other antecedent. If data product A is an antecedent of product B, then A's annotation will contain the endorsement record supporting

its publication, while B's annotation will contain, among other endorsements, a record of the endorsement of A in the context of B.

So far we have discussed the role of context as it relates to the scoping of endorsements. Context also plays a role with respect to the scoping of rules. While the purpose of a rule is to automate endorsement, it is not necessarily appropriate that a rule apply in all cases, i.e., for all users and with respect to all data products. For starters, as mentioned previously, endorsement can only be considered in the context of a particular data product. Additionally, all users might not agree on the endorsement; and, even if they do agree, the desired annotation may differ. Thus the publish tool reads rules from multiple locations: a system-wide configuration file; a configuration file in the user's home directory; and other project- or dataset-specific directories as specified on the command line at invocation time. The storage location of a rule is intended to be indicative of the breadth of its application. A rule placed in a system-wide directory is a statement that all users on that system agree that the rule is appropriate for their work; an example of such a rule might endorse and annotate all files matching `/bin/**/*` as originating from "SunOS 5.10 sparc." There is currently no means of negating or overriding a rule, but it is conceivable that such means might become necessary.

6.2 Difficulties reading provenance graphs

We have found it surprisingly difficult to read and process the provenance graphs produced by ES3, partly due to errors and unanticipated structures in the graphs and partly due to the graphs' sheer complexity. Graph errors result from difficulties in implementing ES3's reconstructive transformation, in which a linear sequence of operating system calls is converted into a directed graph that reflects the provenance of a processing sequence as the publishing scientist might recognize it. These difficulties are exacerbated by an unavoidably imperfect probing system. Specific difficulties encountered include:

- **Granularity limitations**: ES3 determines the connections between files and processes by tracing `open` system calls. But an `open` call doesn't indicate if the file was already existing or not, and the call's modifier flags (in particular, whether the file was opened for read/write access) may obscure the basic relationship between the file and the accessing transformation: Is the transformation reading information from an existing file or writing to a new file? Tracing `read` and `write` calls would resolve the question unambiguously, but this would require logging all I/O performed by the transformation. As a result, the provenance graph may, in some circumstances, contain incorrect or incorrectly oriented edges.
- **Timing limitations**: The ES3 "logger" retrieves information about a referenced file only after the file has been closed. If a file is closed and deleted quickly enough (as is often the case with temporary files), ES3 has no time to obtain information (checksum and last-modified time) about the file. This can lead to missing edges in the provenance graph.

– **Graph representation limitations**: ES3's system call trace log reflects the temporal ordering of a transformation's actions, but the provenance graph, being a simple directed graph, is incapable of representing within-transformation temporality. Thus, a transformation that opens and closes multiple files sequentially will produce a provenance graph that does not and cannot reflect the sequence of those actions, but only that multiple files were read and written. This is a simple consequence of representing transformations and files as nodes in a graph, and of the limited repertoire of edge semantics. As a result, the provenance graph may indicate dependencies that are not warranted.
– **Semantic limitations**: ES3 gathers provenance entirely automatically, relying on no special understanding of the transformation being executed. This lack of higher-level understanding can, in some cases, cause significant misinterpretations of the provenance. For example, consider a program that loops over a set of input files, converting each input file to a distinct output file:

```
for each input file I_k, 1 ≤ k ≤ n:
    read file I_k
    process
    write file O_k
```

In this example, each output file is dependent only on the corresponding input file, and an ideal provenance graph should reflect that. However, ES3, seeing only a sequences of file opens and closes, concludes that output file O_k is dependent on all input files I_j, $1 \leq j \leq k$. Furthermore, because of the fundamental inability to represent temporality in the provenance graph, ES3 represents that every output file is dependent on every input file. Note that the issue here is not that it is impossible for every output file to be dependent on every input file—the issue is that ES3 is unable to detect and represent when that is not the case.
– **Graph complexity**: Files being read and written by transformations represent the most common source of provenance edges, but we have found two other sources of provenance that add to the tracing requirements:
  • **Provenance that traces the derivation of source code files to a binary executable**: A binary executable[5] is represented as a transformation in a processing sequence[6], reading and writing files and interacting with other transformations. But the executable file is itself the target of a separate processing sequence, involving source code compilation or interpretation. Both types of provenance are needed to trace output data files back to the source code that helped produce them.
  • **Provenance between processing sequences**: If a transformation executes another transformation (e.g., if a shell script executes a command), it can pass information through command line arguments and environment variables; these mechanisms are another form of provenance

[5] more precisely, the invocation of the executable within the context of an operating system process
[6] a sequence of processes spawned by a common parent process

to be traced. Thus an ES3 provenance client must trace both "horizontal" edges between transformations and files and "vertical" edges that link transformations to processing sequences, and sequences to parent sequences.

The above-mentioned difficulties can lead to cycles and disconnections in the provenance graph, which are catastrophic to simple graph-walking clients such as the publish tool. And even if the graph does not contain cycles, it may still fail to represent the provenance in a way that reflects the publishing scientist's view of the workflow.

Improvements to ES3 can address some but not all of these issues. Custom system call trace tools might address granularity issues by providing a more accurate view of file-related operations, without having to log all reads and writes. Instrumenting transformations could yield better understanding of within-transformation processing in some cases. More sophisticated processing of ES3's system call trace log might yield more refined and accurate provenance graphs. But the fundamental limitations regarding timing and graph representation remain.

As a result, our overall conclusion is that the publish tool–indeed, any provenance client–needs to be more robust and sophisticated in reading and processing provenance graphs.

6.3 Determining file versions

What is the *version* of a file? We have seen that the publish tool automatically endorses a transformation's input file if the file is managed by a revision control system, in which case the publish tool attempts to annotate the file's endorsement with the file's repository location and revision number. We therefore need to ask the revision control system the question: given a file state (as identified by its pathname and content checksum), what is its revision number? However, revision control systems are designed to answer the converse question: given a revision number of a file, what is the file's content? Thus answering our question requires exhaustively traversing the revision control system's revision graph, looking for file checksum matches.

For revision control systems (such as CVS) that assign revision numbers to individual files, the process is relatively straightforward. Traversing each file's individual history, we identify revisions (there may be more than one) that match the file's checksum at the time recorded in its provenance.

Distributed revision control systems (such as Git[7] and Mercurial) assign revision numbers not to individual files, but to *sets* of changed files ("changesets"). In these systems a changeset reflects the states of *all* files in the repository, even those not changed in that changeset. For example, if a file F is modified in changeset i to have content C, and then in subsequent changeset j F continues to have content C, then both changesets i and j describe file F with content C.

[7] `http://git-scm.com/`

As a consequence it is possible for a file to be described by many, or even all, changesets in a repository.

Whether using file-based or changeset-based revision control, for annotation purposes we would like to select one version identifier as reflecting the file's revision—but which? One option is to use the most recent file revision number or changeset identifier that predates the file reference in the workflow. But a counter-argument can be made to use the earliest changeset that mentions a file (i.e., the changeset at which the file obtained its content as referenced), for changesets are accompanied by commit documentation, and the documentation related to a file will most likely be found in changesets where the file was actually changed.

For the publish tool, our goal is to use the revision number that best describes the referenced file in the sense that the publishing scientist would choose it. Since a repository is in a particular state at the time a transformation executes, and files from that repository are referenced relative to that state, we use the following global approach for each repository encountered. For each referenced file from the repository, we form a set of all changesets that match the state of the file at the time of its provenance reference. We then intersect these sets of changesets. If there is a nonempty intersection, we use the most recent changeset in the intersection that predates the time of transformation execution as the revision number for all referenced files from that repository; if the intersection is empty, we revert to doing the same, but on a file-by-file basis. In any case, if there is a human-assigned tag associated with the identified changeset, we use it as the annotation in preference to the inscrutable checksum-based identifiers that Git and Mercurial use by default. In this way, if a repository was in a particular named state at the time of workflow execution, e.g., "v2.1," then all files referenced from that repository will automatically be endorsed and carry that annotation.

7 Related work

Existing attempts to automate data publication are more focused on enabling data to reproduce specific scientific results. For example, The CDE [6] application virtualizer, like ES3, uses system call tracing to characterize an application's interactions with its environment, but with the goal of packaging the application's executable code, input data, and Linux environment (system libraries, language interpreters, etc.) into a single distributable object, runnable on any Linux system. We are aware of no existing work that specifically uses provenance to automate the data publication process.

Similarly, we are aware of little existing work that actually uses stand-alone provenance to automate a specific task. For example, the Kepler workflow system's Smart Rerun Manager [1] uses provenance to support fine-grained optimization of workflow reruns (e.g., by not re-running processes that provenance has shown to be unaffected by a parameter change), but this occurs in the con-

text of a workflow environment, which preserves much richer semantics than arbitrary collections of processes.

8 Conclusions and further work

The development of ES3 was motivated by a desire to provide provenance of computational workflows similar to what a scientist might specify *a priori*, but to do so *a posteriori* and entirely automatically. The publish tool described here complements ES3's automatic provenance gathering by providing a semi-automated mechanism for mapping ES3 metadata to higher-level descriptions. At the same time, the tool performs integrity checks to ensure that the computation that was actually performed matches the scientist's intention.

As described in section 6.2, further work is required in processing ES3 provenance—indeed, some of the shortcomings identified in ES3 provenance would have been difficult to detect without attempting to use the provenance to drive another computation. Additionally, while the publish tool currently provides some rudimentary hooks for performing other publish-time data management tasks, it is clear that much more work is required in this area.

References

1. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance Collection Support in the Kepler Scientific Workflow System. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, pp. 118-132. Springer, Berlin (2006) doi:10.1007/11890850_14
2. Dozier, J., Frew, J.: Computational provenance in hydrologic science: a snow mapping example. Phil. Trans. R. Soc. A 367(1890), 1021–1033 (2009) doi:10.1098/rsta.2008.0187
3. Frew, J., Metzger, D., Slaughter, P.: Automatic capture and reconstruction of computational provenance. Concurrency Computat.: Pract. Exper. 20, 485–496 (2008) doi:10.1002/cpe.1247
4. Frew, J., Slaughter, P.: ES3: A Demonstration of Transparent Provenance for Scientific Computation. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272, pp. 200–207. Springer, Berlin (2008) doi:10.1007/978-3-540-89965-5_21
5. Frew, J., Janée, G., Slaughter, P.: Provenance-Enabled Automatic Data Publishing. In: Cushing, J., French, J., Bowers, S. (eds.) SSDBM 2011. LNCS, vol. 6809, pp. 244–252. Springer, Berlin (2011) doi:10.1007/978-3-642-22351-8_14
6. Guo, P., Engler, D.: CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In: Proceedings of the 2011 USENIX Annual Technical Conference. (2011) `http://www.usenix.org/event/atc11/tech/final_files/GuoEngler.pdf`
7. Painter, T., Rittger, K., McKenzie, C., Slaughter, P., Davis, R., Dozier, J.: Retrieval of subpixel snow covered area, grain size, and albedo from MODIS. Remote Sensing of Environment 113(4), 868–879 (2009) doi:10.1016/j.rse.2009.01.001