Exploring the future of bioinformatics data sharing and mining with Pygr and Worldbase

Christopher Lee (leec@chem.ucla.edu) – *Department of Chemistry Biochemistry, UCLA, 611 Charles Young Dr. East, Los Angeles, CA 90095* USA
Alexander Alekseyenko (alexander.alekseyenko@nyumc.org) – *Center for Health Informatics and Bioinformatics, Department of Medicine, New York University School of Medicine, New York, NY 10016* USA
C. Titus Brown (ctb@msu.edu) – *Dept. of Computer Science and Engineering, Dept. of Microbiology and Molecular Genetics, Michigan State University, East Lansing, Michigan 48824* USA

Worldbase is a virtual namespace for scientific data sharing that can be accessed via "from pygr import worldbase". Worldbase enables users to access, save and share complex datasets as easily as simply giving a specific name for a commonly-used dataset (e.g. Bio.Seq.Genome.HUMAN.hg17 for draft 17 of the human genome). Worldbase transparently takes care of all issues of how to access the dataset, what code must be imported to use it, what dependencies on other datasets it may have, and how to make use of its relations with other datasets as specified by its schema. Worldbase works with a wide variety of "back-end" storage, including data stored on local file systems, relational databases such as MySQL, remote services via XMLRPC, and "downloadable" resources that can be obtained from the network but automatically installed locally by Worldbase.

Introduction

One of the most important challenges in bioinformatics is a pure Computer Science problem: making it easy for different research groups to access, mine and integrate each other's datasets, in ways that go beyond simple web browsing. The problems of bioinformatics data sharing have grown enormously as the scale and complexity of datasets have increased. Even expert bioinformaticians often find it difficult to work with datasets from outside their specialization; non-expert users often find it impossible. Such difficulties are one of the main barriers to delivering the full value of genomics to all researchers who could benefit from it. Even when good analysis tools are available, the difficulties of accessing and integrating large datasets often limit who can do interesting analyses [LI01].

It doesn't have to be this way. Enabling datasets to "flow" automatically from one place to another, to inter-connect through cross-references that work automatically, and to always bring along the code modules necessary for working with them - these are all infrastructural principles that computer scientists have solved for other domains [DK76]. Good infrastructure should be transparent. Like the Domain Name Service and other technologies that power the web, it should just work, without being visible to the user.

The need for both computational and human scalability

One major obstacle to easy sharing of bioinformatics datasets is their sheer scale and complex interdependencies. Some aspects of this are very simple; for example, many prospective users just don't have enough disk space on their computer to load their desired data. In that case, they should be able to access the data over network protocols transparently; that is, with the exact same interfaces used if the data were stored locally. But a deeper problem is the fact that existing systems for accessing data from other scientists rely on *expertise*: "if the user is an expert in how to use this particular kind of data, s/he'll figure out how to find, download, install, and use this dataset". Since most experts are inexpert at most things, this approach does not scale [BITL]. We have therefore developed a simple but general data-sharing mechanism called `worldbase` [Pygr]:

- To use any public dataset, all the user needs to know is its *name*.
- Asking for a dataset by name yields a full Python interface to it, enabling the user to mine it in all the ways that the original producer of the data could.
- All datasets should be accessible by name from any networked computer, using the closest available resources, or if the user requests, by automatically downloading and installing it locally. Just like the World Wide Web, `worldbase` seeks to create a facade of fully integrated data, not just from the user's computer or LAN but from the whole world.
- The interface for using that data should be exactly the same no matter how or where it is actually being accessed from.
- To use a dataset's relationships to other datasets, again all the user needs to know is the *name* of one of its attributes that links it to another dataset. If the user requests such an attribute, the linked dataset(s) will again be accessed automatically.

In this paper we first describe the current version of `worldbase` (Pygr v.0.8.0, Sept. 2009) by illustrating typical ways of using it. We then discuss some principles of scalable data integration that we believe will prove to be general across many domains of scientific computing.

Using Worldbase

Retrieving a dataset from Worldbase

Say you want to work with human genome draft 18. Start Python and type the following:

```
>>> from pygr import worldbase
>>> hg18 = worldbase.Bio.Seq.Genome.HUMAN.hg18()
```

That's it: you now have the human genome dataset, and can start working. Let's see how many contigs it contains, pull out one chromosome, and print a sequence interval of interest, using standard Python methods:

```
>>> len(hg18)
49
>>> chr1 = hg18['chr1']
>>> len(chr1)
247249719
>>> ival = chr1[20000000:20000500]
>>> print ival
cctcggcctcccaaagtgctgggattacaggcgtgagccaccgcgcagcc...
```

`worldbase` establishes a one-step model for accessing data: *ask for it by name*:

- `worldbase` is an importable namespace for all the world's scientific datasets. You simply *import* the namespace (in the usual Python way), and ask it to *construct* an instance of the dataset that you want (in the usual Python way).
- Of course, this is a *virtual* namespace - you don't actually have all the world's datasets sitting on your computer in a file called `worldbase.py`! `worldbase` connects to a wide variety of data sources (some of which may be on your computer, and some which may be on the Internet) to find out the set of available resources, and then serves them to you.
- `worldbase` takes advantage of Pygr's scalable design. Pygr is first and foremost a *a system of representation* not tied to any fixed assumptions about storage. Pygr is built around the notion of delayed and incremental execution, whereby pieces of a complex and large dataset are loaded only as needed, and in an automatic way that makes this interface transparent. For example, `chr1` represents human chromosome 1, but does not necessarily mean that the human chromosome 1 sequence (245 Mb) was loaded into a Python object). Thus it can work very naturally with huge datasets, even over a network connection where the actual data is stored on a remote server. In this case, `worldbase` is accessing hg18 from UCLA's data server, which is included by default in `worldbase` searches (of course, you can change that).
- To get a dataset, all you need to know is its *name* in `worldbase`. Note that we did not even have to know what code is required to work with that data, let alone explicitly import those modules. `worldbase` takes care of that for you.
- The call syntax (`hg18()`) emphasizes that this acts like a Python constructor: it constructs a Python object for us (in this case, the desired `seqdb.SequenceFileDB` object representing this genome database).
- Note that we did *not* even have to know how to construct the hg18 object, e.g. what Python class to use (`seqdb.SequenceFileDB`), or even to import the necessary modules for constructing it.
- Where did this data actually come from? Since your computer presumably does not contain a local copy of this dataset, `worldbase` accessed it from UCLA's public `worldbase` server over XMLRPC. Currently, our server provides over 460 standard datasets for comparative genomics (such as whole genome sequences, and multigenome alignments), both for client-server access and automated download and installation of datasets (see below).

Storing data in Worldbase

`worldbase` saves not just a data file but a complete Python interface to a dataset, i.e. the capability to *use* and mine the data in whatever ways are possible programmatically. One way of thinking about worldbase is that retrieving data from it is like returning to the moment in time when those data were originally saved to worldbase. Anything you could do with the original data, you can do with the retrieved data.
There are only a few requirements:

- you have your dataset loaded in Python as an object. When retrieved from worldbase, this dataset will be usable by the exact same interface as the original object.
- your object must be **picklable**. Worldbase can store any object that is compatible with standard Python pickling methods. Thus, worldbase is not restricted to Pygr data - but most Pygr classes are of course designed to be stored in worldbase.
- your object must have a docstring, i.e. a `__doc__` attribute. This should give a simple explanatory description so people can understand what this dataset is.

For example, say we want to add the hg17 (release 17 of the human genome sequence) as "Bio.Seq.Genome.HUMAN.hg17" (the choice of name is arbitrary, but it's best to choose a good convention and follow it consistently):

```
from pygr import seqdb, worldbase
# Open the human genome sequence
hg17 = seqdb.SequenceFileDB('hg17')
# Documentation is required to store in worldbase
hg17.__doc__ = 'human genome sequence draft 17'
# Store in worldbase as this name
worldbase.Bio.Seq.Genome.HUMAN.hg17 = hg17
worldbase.commit()
```

Note that you *must* call the function `worldbase.commit()` to complete the transaction and save all pending data resources (i.e. all those added since your last `worldbase.commit()` or `worldbase.rollback()`). In particular, if you have added data to worldbase during a given Python interpreter session, you should always call `worldbase.commit()` or `worldbase.rollback()` prior to exiting from that session.
In any subsequent Python session, we can now access it directly by its worldbase name:

```
from pygr import worldbase
hg17 = worldbase.Bio.Seq.Genome.HUMAN.hg17()
```

You should think of `worldbase` not as a conventional *database* (a container for storing a large set of a specific kind of data) but rather as a *metadata database*, i.e. a container for storing *metadata* describing various datasets (which are typically stored in other, standard databases). By "metadata" we mean information about the *content* of a particular dataset (this is what allows `worldbase` to reload it automatically for the user, without the user having to know what classes to import or how to construct the object correctly), and about its *relations* with other datasets. Throughout this paper, we will use the term "metabase" to refer to this concept of a "metadata database". Whereas a *database* actually stores an entire dataset, a *metabase* merely stores a small amount of metadata pointing to that database and describing its relations with other datasets.

Worldbase automatically captures dataset dependencies

What if you wanted to save a dataset that in turn requires many other datasets? For example, a multigenome alignment dataset is only useful if you also have the genome datasets that it aligns. `worldbase` is smart about figuring out data resource dependencies. For example, you could just save a 17-genome alignment in a single step as follows:

```
from pygr import cnestedlist, worldbase
nlmsa = cnestedlist.NLMSA('/loaner/ucsc17')
nlmsa.__doc__ = 'UCSC 17way multiz alignment, on hg17'
worldbase.Bio.MSA.UCSC.hg17_multiz17way = nlmsa
worldbase.commit()
```

This works, even though using this 17-genome alignment (behind the scenes) involves accessing 17 `seqdb.SequenceFileDB` sequence databases (one for each of the genomes in the alignment). Because the alignment object (NLMSA) references the 17 `seqdb.SequenceFileDB` databases, worldbase automatically saves information about how to access them too.
However, it would be a lot smarter to give those databases worldbase resource names too:

```
from pygr import cnestedlist, worldbase
nlmsa = cnestedlist.NLMSA('/loaner/ucsc17')
for resID, genome in nlmsa.seqDict.prefixDict.items():
  # 1st save the genomes
  genome.__doc__ = 'genome sequence ' + resID
  worldbase.add_resource('Bio.Seq.Genome.' + resID,
                         genome)
nlmsa.__doc__ = 'UCSC 17way multiz alignment, on hg17'
# now save the alignment
worldbase.MSA.Bio.UCSC.hg17_multiz17way = nlmsa
worldbase.commit()
```

This has several advantages. First, we can now access other genome databases using worldbase too:

```
from pygr import worldbase
# get the mouse genome
mm7 = worldbase.Bio.Seq.Genome.mm7()
```

But more importantly, when we try to load the ucsc17 alignment on another machine, if the genome databases are not in the same directory as on our original machine, the first method above would fail, whereas in the second approach worldbase now will automatically figure out how to load each of the genomes on that machine.

Worldbase schema automatically connects datasets for you

One major advantage of `worldbase` is that it explicitly captures and automatically applies schema information about relationships and interconnections between different datasets. By "schema" we mean the precise connections between two or more collections of data. Such inter-relations are vital for understanding and mining scientific datasets. For example "a genome has genes, and genes have exons", or "an exon is connected to another exon by a splice". Let's say we have two databases from `worldbase`, exons and splices, representing exons in a genome, and splices that connect them, and a mapping splicegraph that stores the many-to-many connections between exons (via splices). We can add splicegraph to `worldbase` and more importantly save its schema information:

```
splicegraph.__doc__ = 'graph of exon:splice:exon links'
worldbase.Bio.Genomics.ASAP2.hg17.splicegraph = splicegraph
worldbase.schema.Bio.Genomics.ASAP2.hg17.splicegraph = \
  metabase.ManyToManyRelation(exons, exons, splices,
                bindAttrs=('next','previous','exons'))
worldbase.commit()
```

This tells `worldbase` that splicegraph is a many-to-many mapping from the exons database onto itself, with "edge" information about each such mapping stored in the `splices` database. Concretely, this means that for each `exon1` to `exon2` connection `splice`, then `splicegraph[exon1][exon2]=splice`. Furthermore, the `bindAttrs` option says that we wish to bind this schema as named attributes to all items in those databases. Concretely, for any object `exon1` from the `exons` database, it makes `exon1.next` equivalent to `splicegraph[exon1]`. That means a user can find all the exons that `exon1` splices to, by simply typing:


```
for exon2,splice in exon1.next.items():
   do something...
```

Note how much this simplifies the user's task. The user doesn't even need to know about (or understand) the `splicegraph` database, nor indeed do anything to load `splicegraph` or its dependency splices. Consistent with the general philosophy of `worldbase`, to use all this, the user only needs to know the name of the relevant attribute (i.e. exons have a `next` attribute that shows you their splicing to downstream exons). Because `worldbase` knows the explicit schema of `splicegraph`, it can automatically load splicegraph and correctly apply it, whenever the user attempts to access the `next` attribute. Note also that neither splicegraph nor splices will actually be loaded (nor will the Python module(s) need to construct them be loaded), unless the user specifically accesses the `next` attribute.

Controlling where Worldbase searches and saves data

`worldbase` checks the environment variable `WORLDBASEPATH` for a list of locations to search; but if it's not set, worldbase defaults to the following path:

```
~,.,http://biodb2.bioinformatics.ucla.edu:5000
```

which specifies three locations to be searched (in order): your home directory; your current directory; the UCLA public XMLRPC server. Worldbase currently supports three ways of storing metabases: in a Python shelve file stored on-disk; in a MySQL database table (this is used for any metabase path that begins with "mysql:"); in an XMLRPC server (this is used for any metabase path that begins with "http:").

Worldbase can install datasets for you locally

What if you want to make `worldbase` download the data locally, so that you could perform heavy-duty analysis on them? The examples above all accessed the data via a client-server (XMLRPC) connection, without downloading all the data to our computer. But if you want the data downloaded to your computer, all you have to do is add the flag `download=True`. For example, to download and install the entire yeast genome in one step:

```
yeast = \
  worldbase.Bio.Seq.Genome.YEAST.sacCer1(download=True)
```

We can start using it right away, because `worldbase` automates several steps:

- `worldbase` first checked your local resource lists to see if this resource was available locally. Failing that, it obtained the resource from the remote server, which basically tells it how to download the data.
- `worldbase` unpickled the `Bio.Seq.Genome.YEAST.sacCer1` `seqdb.SequenceFileDB` object, which in turn requested the `Bio.Seq.Genome.YEAST.sacCer1.fasta` text file (again with `download=True`).
- this is a general principle. If you request a resource with `download=True`, and it in turn depends on other resources, they will also be requested with `download=True`. I.e. they will each either be obtained locally, or downloaded automatically. So if you requested the 44 genome alignment dataset, this could result in up to 45 downloads (the alignment itself plus the 44 genome sequence datasets).
- the compressed file was downloaded and unzipped.
- the `seqdb.SequenceFileDB` object initialized itself, building its indexes on disk.
- `worldbase` then saved this local resource to your local worldbase index (on disk), so that when you request this resource in the future, it will simply use the local resource instead of either accessing it over a network (the slow client-server model) or downloading it over again.

Some scalability principles of data Integration

We and many others have used `worldbase` heavily since its release (in Pygr 0.7, Sept. 2007). For some examples of large-scale analyses based on `worldbase` and Pygr, see [AKL07] [Kim07] [ALS08].

We have found `worldbase` to be a work pattern that scales easily, because it enables any number of people, anywhere, to use with zero effort a dataset constructed by an expert via a one-time effort. Furthermore, it provides an easy way (using `worldbase` schema bindings) to integrate many such datasets each contributed by different individuals; these schema relations can be contributed by yet other individuals. In the same way that many software distribution and dependency systems (such as CRAN or fink) package code, `worldbase` manages data and their schema. Attempts to encapsulate data into a global distribution system are made in CRAN [CRAN], but limited to example datasets for demonstrating the packaged code. Unlike CRAN, `worldbase`'s focus is primarily on data sharing and integration.

Based on these experiences we identify several principles that we believe are generally important for scalable data integration and sharing in scientific computing:

- **Encapsulated persistence:** we coin this term to mean that saving or retrieving a dataset requires nothing more than its *name* in a virtual namespace. Moreover, its dependencies should be saved / retrieved automatically by the same principle, i.e.

simply by using their *names*. This simple principle makes possible innumerable additional layers of automation, because they can all rely on obtaining a given dataset simply requesting its name.

- **Interface means representation, not storage:** the public interface for working with `worldbase` datasets must provide a powerful representation of its data types, that can work transparently with many different back-end storage protocols. For example, the back-end might be an XMLRPC client-server connection across a network, or a MySQL database, or specially indexed files on local disk. By using this transparent interface, user application code will work with any back-end; moreover, users should be able to request the kind of back-end scalability they want trivially (e.g. by specifying `download=True`).

- **Scalability is paramount:** in all of our work, scalability has been a primary driving concern - both computational scalability and human scalability. Scientists will only mine massive datasets using a high-level language like Python (whose high-level capabilities made `worldbase` possible) when this retains high performance. Fortunately, elegant solutions such as Pyrex [Pyrex] and Cython [Cython] make this entirely feasible, by enabling those components that are truly crucial for performance to be coded in optimized C.

- **Metabases are the glue:** we coin the term "metabase" to mean a *metadata database*. `worldbase` is *not* intended to be a database in which you actually store data. Instead it is only intended to store *metadata* about data that is stored elsewhere (in disk files; in SQL databases; in network servers, etc.). Broadly speaking these metadata for each resource include: what kind of data it is; how to access it; its relations with other data (schema and dependencies). Metabases are the heart of `worldbase`.

- **Provide multiple back-ends for 3 standard scalability patterns:** Users most often face three different types of scalability needs: **I/O-bound**, data should be worked with in-memory to the extent possible; **memory-bound**, data should be kept on-disk to the extent possible; **CPU-bound** or **disk-space bound**, data should be accessed remotely via a client-server protocol, as there is either no benefit or no space for storing them locally. As an example, `worldbase` has been used so far mainly with Pygr, a framework of bioinformatics interfaces and back-ends. For each of its application categories (e.g. sequences; alignments; annotations), Pygr provides all three kinds of back-ends, with identical interfaces.

- **Standard data models: containers and mappings**. In Pygr and `worldbase`, data divide into two categories: *containers* (a dictionary interface whose keys are identifiers and whose values are the data objects) and *mappings* that map items from one container (dataset) onto another. In particular, Pygr (the Python Graph database framework) generalizes from simple Python mappings (which store a one-to-one relation) to graphs (many-to-many relations). By following completely standard Python interfaces [Python] for containers, mappings and graphs (and again providing three different kinds of back-ends for each, to cover all the usual scalability patterns), Pygr makes `worldbase`'s simple schema and dependency capabilities quite general and powerful. For example, since Pygr's mapping classes support Python `__invert__()` [Python], `worldbase` can automatically bind schema relations both forwards and backwards.

- **Schema is explicit and dynamic:** We have defined *schema* as the metadata that describe the connections between different datasets. When schema information is not available as data that can be searched, transmitted, and analyzed at run-time, programmers are forced either to hard-wire schema assumptions into their code, or write complex rules for attempting to guess the schema of data at runtime. These are one-way tickets to Coding Hell. `worldbase` is able to solve many data-sharing problems automatically, because it stores and uses the schema as a dynamic graph structure. For example, new schema relations can be added between existing datasets at any time, simply by adding new mappings.

- **The web of data interconnects transparently:** These schema bindings make it possible for data to *appear* to interconnect as one seamless "virtual database" (in which all relevant connections are available simply by asking for the named attributes that connect to them), when in reality the datasets are stored separately, accessed via many different protocols, and only retrieved when user code specifically requests one of these linked attributes. This can give us the best of both worlds: an interface that looks transparent, built on top of back-ends that are modular. In fact, one could argue that this principle is the programmatic interface analogue of the hyperlink principle that drove the success of the hypertext web: a facade of completely inter-connected data, thanks to a transparent interface to independent back-ends. From this point of view one might consider `worldbase` to be a logical extension of the "semantic web" [BHL01], but reoriented towards the scalability challenges of massive scientific computing datasets. We think this transparent interconnection of data is gradually becoming a general principle in scientific computing. For example, the Comprehensive R Archive Network like `worldbase` provides a uniform environment for accessing packages of code + data that can be installed with automatic links to their dependencies, albeit with a very different interface reflecting the limitations of its language en-

vironment (R instead of Python).

Current limitations and plan for future development

- Currently, `worldbase` supplies no mechanism for global "name resolution" analogous to the DNS. Instead, the user simply designates a list of `worldbase` servers to query via the `WORLDBASEPATH` environment variable; if not specified it defaults to include the main (UCLA) `worldbase` server. This simple approach lets users easily control where they will access data from; for example, a user will typically give higher precedence to local data sources, so that `worldbase` requests will be obtained locally if possible (rather than from a remote server). This lack of centralization is both a vice and a virtue. On the one hand users are free to develop and utilize resources that best suit their research needs. On the other hand, such lack of centralization may often result in duplication of effort, where two research would work on the same data transformation without knowing of each other's efforts. We believe that these problems should be solved by a centralized mechanism similar to the DNS, i.e. that enables data producers to publish data within "domains" in the name space that they "own", and transparently resolves name requests to the "closest" location that provides it.
- Since `worldbase` uses Python pickling, the well-known security concerns about Python unpickling also apply to `worldbase`. These must be resolved prior to expanding `worldbase` from a user-supplied "access list" to a DNS-like global service. We believe that in a public setting, pickle data should be authenticated by secure digital signatures and networks of trust using widely deployed standards such as GnuPG [GnuPG].

Future developments:

- Support for novel and emerging data types, for example:
  - Genome-wide association study (GWAS) data.
  - Next-generation sequencing datasets, such as RNA-seq, allele specific variation, ChIP-seq, and microbiomic diversity data.
- Increased support for using `worldbase` within common cluster computing systems. This seems like a natural way of being able to seamlessly scale up an analysis from initial prototyping to large-scale cluster computations (very different environments where often data resources cannot be accessed in exactly the same way), by pushing all data access issues into a highly modular solution such as `worldbase`.
- Optimized graph queries.
- Data visualization techniques.
- Ability to push code objects along with the data, so that class hierarchy and appropriate access methods may be installed on the fly. In the context of digitally signed code and networks of trust, this could greatly increase the convenience and ease with which scientists can explore common public datasets.

Acknowledgments

We wish to thank the Pygr Development team, including Marek Szuba, Namshin Kim, Istvan Alberts, Jenny Qian and others, as well as the many valuable contributions of the Pygr user community. We are grateful to the SciPy09 organizers, and to the Google Summer of Code, which has supported 3 summer students working on the Pygr project. This work was also supported from grants from the National Institutes of Health (U54 RR021813; SIB training grant GM008185 from NIGMS), and the Department of Energy (DE-FC02-02ER63421).

References

[LI01] C. Lee, K. Irizarry, *The GeneMine system for genome/proteome annotation and collaborative data-mining. IBM Systems Journal* 40: 592-603, 2001.

[DK76] F. Deremer and H.H. Kron, *Programming In the Large Versus Programming In the Small. IEEE Transactions On Software Engineering*, 2(2), pp. 80-86, June 1976.

[BITL] D.S. Parker, M.M. Gorlick, C. Lee, *Evolving from Bioinformatics in the Small to Bioinformatics in the Large. OMICS*, 7, 34-48, 2003.

[Pygr] The Pygr Consortium, *Pygr: the Python Graph Database Framework*, 2009, http://pygr.org.

[AKL07] A.V. Alekseyenko, N. Kim, C.J. Lee, *Global analysis of exon creation vs. loss, and the role of alternative splicing, in 17 vertebrate genomes. RNA* 13:661-670, 2007.

[Kim07] N. Kim, A.V. Alekseyenko, M. Roy, C.J. Lee, *The ASAP II database: analysis and comparative genomics of alternative splicing in 15 animal species. Nucl. Acids Res.* 35: D93-D98, 2007.

[ALS08] A.V. Alekseyenko, C.J. Lee, M.A. Suchard, *Wagner and Dollo: a stochastic duet by composing two parsimonious solos. Systematic Biology* 57: 772-784, 2008.

[CRAN] *The Comprehensive R Archive Network*, from http://cran.r-project.org/.

[Pyrex] G. Ewing, *Pyrex - a Language for Writing Python Extension Modules.* http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex.

[Cython] S. Behnel, R. Bradshaw, D.S. Seljebotn, *Cython: C Extensions for Python.* http://www.cython.org.

[Python] G. van Rossum and F.L. Drake, Jr., *Python Tutorial*, 2009, http://www.python.org.

[BHL01] T. Berners-Lee, J. Hendler, O. Lassila, *The Semantic Web. Sci. Am.* May 2001.

[GnuPG] *GNU Privacy Guard*, http://www.gnupg.org/