



1.1 Perl based framework for distributed processing – by Douglas White & John Tebbutt

Software Diagnostics and Conformance Testing Division
National Institute of Standards and Technology
Email: nsrl@nist.gov

1.1.1 Abstract

The National Software Reference Library (NSRL) of the U.S. National Institute of Standards and Technology (NIST) collects software from various sources and publishes file profiles computed from this software (such as MD5 and SHA-1 hashes) as a Reference Data Set (RDS) of information. The RDS can be used in the forensic examination of file systems, for example, to speed the process of identifying unknown or suspicious files.

This paper describes the cross-platform, public domain, Linux/Apache/MySQL/Perl (LAMP) framework with which we produce the RDS from acquired software. The framework is easily deployed (it has been packaged on a Knoppix-based live CD) and allows for the distributed processing of large numbers of files in a loose, heterogeneous computing cluster. We go on to suggest that the framework is sufficiently general in its implementation to be suitable for application to classes of problems quite beyond our original scope.

1.1.2 Introduction

1.1.2.1 Overview of the NSRL

The National Software Reference Library (NSRL) [1] provides a set of reference data (the NSRL Reference Data Set, or RDS) that can be used to identify computer files. This identification may be used to target specific files of interest or to eliminate "known good" files from further review during a digital forensic investigation. The ability to rapidly identify files in this way can significantly decrease the amount of time required, for example, to gather and verify evidence in crimes involving computers.

The NSRL collects software from various sources, primarily through the purchase of shrinkwrapped physical media, and also through donations from vendors and other parties. This software is recorded as the original source for known files and stored as a permanent part of the NSRL. Our aim is to collect as many different examples, versions, and updates of software as possible in order to generate file signatures for as many known files as possible.

The NSRL RDS is built on file signature generation technology that is used primarily in cryptography. The selection of the specific file signature generation routines is based on customer requirements and the necessity to provide a level of confidence in the reference data that will allow it to be used in the U.S. Courts. A white paper [2] gives an overview of the various hashing algorithms considered, as well as implementations of those algorithms. It also gives factors regarding their selection and use. At this time, the algorithms in use are the 32-bit cyclic redundancy code (CRC32), message digest level 4 (MD4), message digest level 5 (MD5) and the Secure Hashing Algorithm revision 1 (SHA-1).

The September 2004 Version 2.6 release of the RDS contains the file signatures and other metadata from the files contained in 5,902 software products. The NSRL project has processed over 45 million files to this point, equating to more than 3 terabytes of data. Due to duplication of files in the software packages, the RDS holds the information on 28,259,959 unique files or the more precise measure of 9,913,096 unique SHA-1 file signatures from the 45 million file collection.

1.1.2.2 The NSRL distributed processing framework

This paper describes our use of Perl to automate the harvesting of file signatures from software package media in a way which ensures the highest degree of correlation between the set of files contained on the media and the set of files which is transferred onto working systems during the package installation process.



We believe that the method we describe for hashing a library of files is sufficiently generic that it may be used, with only minor modifications, in a wide range of applications. For example, code unraveling, decryption, signal processing, etc. Law enforcement, digital archival, corporate security and forensic investigation applications can also be developed within this framework.

1.1.3 Problem Domain

The mission of the NSRL is to provide reference data which can be used to unambiguously identify installed files on an arbitrary computer. The data we supply are in the form of hashes of files included in software distributions - for each file, we include identification information for the file (the name of the package from which the file originated, the manufacturer of the package, etc.) and the SHA-1, MD5, MD4 and CRC32 hashes of the file.

Several factors make it impractical to gather this data by actually installing packages on running systems, including: the enormous range and diversity of available software; multiple versions of any given package; and variations in installation techniques between vendors and packages. Most importantly, a paramount concern for us is reproducibility and traceability of the data we provide: we must be able, at any time, to produce a genuine software distribution package and demonstrate that a file identified on a system using our data actually originated from that package. As a result, we rely on the processing of files directly from installation media.

1.1.4 Motivation to Create a Public Domain Framework

1.1.4.1 History

The first version of our hashing framework served us quite well in our core mission - the production of reference data sets for file identification. In addition, we started to receive requests from other organizations, either to process batches of files on their behalf or to build (or provide expertise or a blueprint to build) similar installations at their facilities. Such requests presented us with an excellent opportunity to expand the scope of the framework; however, various obstacles prevented us from doing so:

We are not permitted to handle certain kinds of material at our facility (e.g. illegal or otherwise illicit material; material relating to national security - domestic or foreign).

Resource constraints prevent us from providing a custom dataset service. Such activity would have the potential to detract significantly from our core mission and we would be unable to guarantee the necessary quality of service.

Similarly, we lack the resources either to recreate (and subsequently support) bespoke versions of the framework for interested parties or to produce the extensive documentation that would be required in order to enable non-technical parties to build and support a system themselves. Either activity would detract significantly from our core mission.

Our implementation of the framework was based around a number of dedicated, fairly high-end systems. Two high performance server platforms were chosen for hosting the database and the shared file space, using large RAID storage devices, fifteen cutting edge desktop computers were chosen to perform the mathematical hashing algorithms, and six older systems were used as data entry stations. Establishment and maintenance of such a dedicated architecture might be beyond the expertise and budgetary constraints of many organizations that might benefit from it.

The framework was constructed around a proprietary codebase, so any attempt to recreate it would require the purchase of various software licenses by organizations, some of whom could ill afford to do so.

Finally, we needed to broaden the scope of the software we could process. Because the framework was built on a single, proprietary, platform, at least some material designed for use on other platforms would always be beyond our reach. Our goal was to transition to a multi-platform architecture in which heterogeneous nodes cooperate to process material intended for different



platforms.

1.1.4.2 Solution

Our approach was to redesign the framework in accordance with three central principles:

Make the framework available as a one-stop package, including all necessary software: refactor the codebase to require only free or open source software; use LAMP tools; rely on Perl for platform neutral operation.

Package the codebase such that installation, configuration and operation of the framework require minimal effort or expertise: provide all necessary packages and perform all necessary bootstrapping on installation.

Produce a distribution which has no requirement for dedicated hardware: develop a Linux-based Live CD for Intel/compatible systems to enable ad hoc creation of a processing environment using any available machines.

The use of open source (LAMP) software was not a point of contention with the project sponsors. In fact, the Linux-based Knoppix Live CD [3] and derivatives (see, for example [4], [5] and [6]) have a history of use in the digital forensics community, and a corresponding level of familiarity and expertise with open source tools exists.

Adherence to these principles has yielded a framework which is platform neutral, easily deployed, resource independent and freely available. We have increased the scope of our own operation and produced a system that can be quickly deployed and easily modified by interested parties.

1.1.5 System Architecture

Conceptually, the system is made up of three types of node: batching nodes, processing nodes and database servers which operate in concert in a loosely coupled cluster. In practice, any or all of these node types may co-reside on a single system: for the sake of speed of processing and data throughput, our configuration currently comprises dedicated systems housing a single database server and variable numbers of batching nodes and processing nodes, as shown in Figure 1.

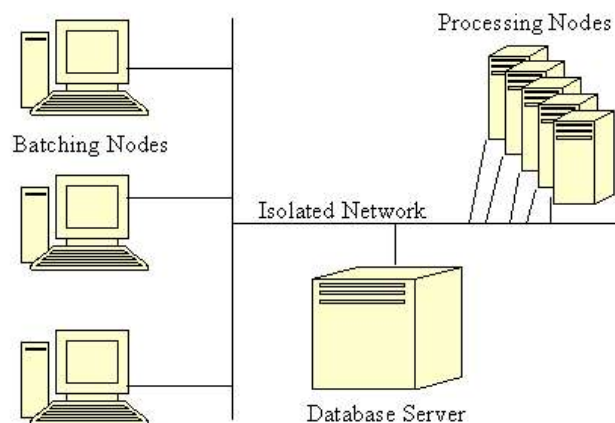


Figure 1. Example system architecture of an NSRL framework deployment.

Because of the sensitivity of our data and a requirement for high throughput, our systems communicate via a dedicated, isolated ethernet LAN.

1.1.5.1 Batching nodes

For our purposes, the batching nodes represent the point of entry of data into the system. A human operator enters metadata about a specific piece of software distribution media (floppy, CD or DVD), including a unique identifier, manufacturer and package name, at the batching node system via a web interface to the database server. The piece of media is then placed into the appropriate device



on the batching node system, this device is remotely mounted by the database server via SMB, and its contents copied across the network and stored along with its identifying information. Once the contents of the media have been copied, the batching system is ready to accept the next piece of media.

Since the role of the batching node is relatively simple, all that is required of such a node is that it have the capability to act as an SMB client and run a reasonably functional web browser.

We are investigating methods to automatically identify each piece of media, with the intent of minimizing human involvement in metadata generation, thereby removing a potential source of error and/or inconsistency in package metadata. It would also relieve numerous colleagues of a great deal of tedium (!).

1.1.5.2 Database Server nodes

The database server has three principal functions:

- It directs the acquisition and storage of raw data and associated metadata through a web interface to the batching node(s), as described above;

- It directs the processing of data by maintaining an inventory of data for processing and its location, assigning data to processing nodes, and tracking these assignments;

- It stores the results of data processing, which can be output periodically in a format suitable for distribution.

1.1.5.3 Processing nodes

The processing nodes are responsible for processing the raw data to produce results. In the case of the NSRL cluster, the raw data is the collection of files which make up the contents of a piece of distribution media, while the results are a collection of hashes and checksums derived from those files.

Each processing node polls the database server for work. If none is available, it sleeps for a period before polling again. If work is available, it gets the location of the data from the database server and begins processing. Results are stored locally until processing is complete, at which point the node passes the results to the database server and polls for more work.

1.1.5.4 NOTES

Because batching nodes and processing nodes operate in the role of clients to the database server, the system as a whole is resilient to the comings and goings of either type of client. This means that the overall size of the configuration can be altered dynamically to handle changing workloads.

Batching and processing nodes can be added to the configuration and be integrated at any time. They may also be removed dynamically at breakpoints, which occur between jobs. Removal of nodes between breakpoints does not cause system failure, however some manual intervention is required to guard against loss of data.

1.1.6 A Modular Approach

1.1.6.1 CPAN modules

We use Perl almost exclusively as our development language because of its cross-platform portability. As a result, we have made extensive use of CPAN modules, including:

Hashing:

- Digest::SHA, Digest::MD4, Digest::MD5, String::CRC32 - these modules provide all the functionality we need for the generation of file signatures.



File classification:

File::MMagic - used for identification of archive files which must be recursively extracted before their contents can be processed.

Configuration:

Tiny::Config - used to process configuration files.

Logging:

Log::Dispatch - used for system-wide activity logging.

1.1.6.2 Beyond CPAN

There are three areas in which we have found it desirable to improve upon or enhance the functionality provided by existing CPAN modules.

1.1.6.2.1 Database abstraction

The NSRL codebase includes a database abstraction module [7] which serves as a framework for both object and relationship persistence via a SQL capable database. Interaction with an underlying database is accomplished via Perl objects that abstract away the construction, issuance, and processing of SQL queries. Using this module, it is possible to create, modify and delete objects, as well as link them together in arbitrary ways, and perform a variety of querying operations. The querying engine is reasonably sophisticated, allowing for a large degree of flexibility without becoming overly cumbersome, and performs intelligent buffering of results so as to be extensible to large queries.

The module isolates the framework from dependence on a particular database application for persistent data storage, and will ease our transition to a web services-based distribution model for the NSRL reference data.

We hope this module will be of general interest, and are working to add it to CPAN.

A more detailed description of the module is included in Appendix 1.

1.1.6.2.2 CRC32 checksum calculation

We discovered that String::CRC32 was taking an order of magnitude more time to calculate hashes than the other hashing modules we were using. This led to a significant decrease in performance when compared with the proprietary-based system we were replacing.

We tried the CPAN modules String::CRC::Cksum and Digest::CRC, only to find that these were an order of magnitude again slower than String::CRC32 (see Table 1).

String::CRC32 uses inlined C code to perform its calculations. On a hunch that perhaps the algorithm used in the C code was responsible for the relatively poor performance of the module, we wrote our own CRC32 module, in which we inlined an updated variant of the C code used on our old system, which we obtained from the SNIPPETS C Source Code Archives [8].

Note: The SNIPPETS archive is a collection of C code fragments, each of which is ``either explicitly in the public domain or which carries a free use license with few or no restrictions which would prevent its use''. The code fragments are also said to be compilable across multiple platforms.

The new module had approximately the same performance as String::CRC32. As a final test of the algorithm, we compiled the C code and ran it from the system using Perl backticks. Performance was hugely improved, to the extent that the CRC32 hash was now the fastest of the hashes performed - exactly as one would expect.



Table 1 shows performance comparisons for the methods described, with SHA-1 hash calculations using Digest::SHA for comparison.

Algorithm	Time (secs)
String::CRC::Checksum	70
Digest::CRC	43
String::CRC32	6
Precompiled CRC32	0.02
Digest::SHA (SHA-1)	0.5

Table 1. Approximate time in seconds to calculate hashes of 20 1MB files on a 1.4GHz Pentium 4 system with 1GB RAM (SHA-1 calculation time given for comparison).

1.1.6.2.3 Cracking archives

Our research has shown that, when software is installed onto a computer, some 60% - 80% of the installed files are identical to files contained on the installation media (CDROMs, DVDROMs, downloads, floppy disks, etc.). However, the installed files often are not merely copied from the media: many files are contained in archives, and are extracted and placed onto the target machine during the installation process. Thus it is not sufficient to merely hash every file on the distribution media: we also need to identify any archives and (recursively) extract and hash the files they contain. In this sense, we need to mimic the installation process in order to get the maximum information from the distribution media.

Opening the many, often proprietary, archive formats present on software distribution media has proven to be a challenge. CPAN contains a relatively small number of archiving utilities, the majority of which seem to be wrappers for OS-specific libraries (examples include Compress::Bzip2, Compress::Zlib). Thus to maximize the utility of the framework, it is necessary to use platform-specific tools. How we accomplish this while maintaining platform neutrality is described below.

1.1.7 Inter-node communication and cooperation

The NSRL framework uses platform-specific binaries to open archives recursively. This section describes how this is done, specifically:

Within a processing node: how can the framework assess whether it is able to open an archive and, if it is, which external binary to call?

Between processing and database nodes: what should a node do if it determines it cannot open an archive?

1.1.7.1 Finding the right application

A processing node determines the type of a file archive using the CPAN module File::MMagic in conjunction with a custom magic file. The node then uses a combination of its operating system type and the archive type to look up the appropriate native application for processing of the archive in a configuration table. This process was previously described in [9].

1.1.7.2 Cooperative processing

The processing nodes communicate with the database node (s) using standard ODBC calls. The database server maintains a dynamic database of work items to be processed. A database record



for a work item consists of the following information:

Item ID

A unique identifier for the work item. Actually a composite of several identifiers which uniquely identifies a piece of installation media within a package;

Item Location

The location of the batched contents of a piece of installation media;

Upload flag

A flag indicating whether the item is ready for processing (i.e. has been completely uploaded from a batching node);

Claimed flag

A flag indicating whether the item has been claimed by a processing node (i.e. whether a processing node is currently working on the item - effectively a locking flag);

Claiming Node

The ID of the node that is currently processing the item;

Completed flag

A flag indicating whether processing of an item is complete;

OS Hint

A hint as to the operating system a node should have in order to best process this item.

The processing nodes poll the work item database for work items that are uploaded, not claimed and not completed. If an item is found, the node sets the Claimed flag, sets the Claiming Node field to its own ID, and copies the item from the specified location to its local file store.

In the case where the node is able to process any and all archives contained in the item, once it has completed processing, it inserts its results into the results database, sets the Completed flag in the work item database, and queries the work item database for more work.

If a processing node finds one or more archives which it unable to open, it processes them as normal (flat) files and continues processing the item until it is done. Upon completion, it checks the results database: if results exist for files below the level of all archives it was unable to open, then processing of the item is complete. The node inserts its results into the results database, sets the Completed flag in the work item database, and queries the work item database for more work.

If the processing node sees that no results exist for files below the level of one or more archives it was unable to open, it inserts its results into the results database then returns the item to the work pool: it unsets the Claimed flag in the work item database, does not set the Completed flag, and may insert a value into the OS Hint field if its configuration file contains information linking one or more of the archive types to a specific OS.

1.1.7.3 An example

Suppose we have an collection of processing nodes which includes Windows, Linux and OS X systems, and an installation CD for a software package which contains versions for all three platforms in the form of an InstallShield .exe file, a Stuffit archive and an RPM file.

The first processing node to detect the work item once it is uploaded is a Linux system. It marks the



item as claimed, inserts its system ID, and copies the item to its local filesystem. The node will detect all three archive files, but we'll assume it can only open and process the contents of the RPM file: the .exe and the .sit archives are processed as flat files. The node sees that no results exist in the results database for files below the level of the unprocessable archives, so it posts its results to the results database, and unsets the Claimed flag for the work item. It may also set the OS Hint field for the item to Windows or OS X, depending on its configuration and the order in which it encounters the archives.

The next node to detect the work item is a Windows system. It marks the item as claimed, inserts its system ID, and copies the item to its local filesystem. The node will detect all three archive files, but can only open and process the contents of the InstallShield file: the RPM and Stuffit archives are processed as flat files. The node sees that no results exist in the results database for files below the level of the Stuffit archive, so it posts its results to the results database, and unsets the Claimed flag for the work item. It may also set the OS Hint field for the item to OS X, depending on its configuration.

Finally, an OS X node detects the work item, and is able only to open the Stuffit archive. However, upon completion of the item, it sees in the results database that the RPM and InstallShield archives have results posted for files below them, which indicates that processing of the item is complete. The node posts its results to the results database and sets the Completed flag in the work item database.

While this example is simplified for the sake of brevity, it nevertheless serves to illustrate two central capabilities of the framework:

The framework can process any file, regardless of its provenance, so long as a Perl interpreter exists for the operating system to which the file is bound;

The framework can incorporate any system for which a Perl interpreter exists.

As a result, the framework is suitable for deployment for a wide range of tasks under a wide range of conditions.

Heterogeneous systems

The processing nodes can be any mix of architectures and operating systems which, as is the case for the NSRL, enables a single framework system to incorporate ``specialist'' nodes to handle tasks they are better suited to.

Low-cost systems

The framework can incorporate large numbers of heterogeneous systems, so that a framework system can be assembled at little or no cost from older or surplus machines. There is no requirement to match the architectures, operating systems or processing power of the constituent nodes, since all of these features can be accounted for in a global configuration file which is installed on each system as it joins the framework.

Replace archive unpacking with your favorite app

Perhaps most significantly for the broader community, the framework can be easily adapted to deploy a distributed processing model to any task involving large file manipulation. For example, if the custom magic file were tailored to recognize MP3-encoded audio files, and the processor node configuration file were configured to run a steganography detection application on such files, the framework would become a distributed search engine for steganographically encoded information in large file systems.

1.1.8 Performance



The isolated NSRL subnet runs at a minimum 100 Mb/s network speed, with some hubs and computers capable of 1 Gb/s speed, as we prepare for an upgrade.

The database server is a dual 800 MHz CPU server with 4GB of RAM and a 160 GB RAID-5 disk array. The file server is a 2 GHz CPU server with 2GB of RAM and a 180 GB RAID-5 disk array.

The hashing nodes are comprised of two different hardware platforms. One configuration uses an Intel 2GHz CPU with 4GB of RAM and 80 GB of disk space; twelve such systems are used, running either Microsoft Windows or Linux, along with the hashing code and third party applications. The other hardware configuration uses a dual 2GHz G5 CPU with 8GB RAM and 250 GB of disk space; three such systems are used, running Apple Mac OS X, along with the hashing code and third party applications.

One 400 MHz CPU with 512MB RAM running Windows NT 4 serves as the current domain controller. One 800 MHz CPU with 512 MB RAM running Linux serves as the current DHCP server. One 2 GHz CPU with 2 GB RAM running Linux serves as the current firewall and router when the network airgap is closed. Six CPUs of various unremarkable configurations run Microsoft Windows 98 and serve as the batching stations.

As an example of the system throughput, we harvested 250,271 SHA-1 hashes from the MSDN Disc2386 DVD (RDS application # 5949) in the period of time from 2004-07-20 10:50:23 to 2004-07-20 22:20:16, which is 11 hours, 29 minutes and 53 seconds. This is a rate of 21,766 files per hour. The amount of data processed was over 33,733 MB, or roughly 2,933 MB per hour using only one of the fifteen hashing nodes. We hashed 33 GB from that one DVD due to the amount of compressed and archived files present. This gives an average throughput of 814 KB/second, including byte signature check, unarchiving, and application of four hash algorithms to each file. This figure represents the typical performance of our system.

1.1.9 Live CD Distribution

The NSRL framework has been incorporated into a prototype, bootable CD based on the popular Knoppix live Linux CD distribution [3].

1.1.9.1 Motivation

Over its history, the NSRL project has received various requests and enquiries regarding processing of non-NSRL material and the use of clusters built on the NSRL processing framework. We needed to make the framework available and accessible to even the most resource-constrained organizations. Packaging the framework along with a bootable operating system environment provides a one-stop, low- or zero-cost method to enable interested parties to set up their own processing frameworks with minimal expertise and minimal resource requirements. The bootable CD distribution can be used to set up processing clusters, on either a permanent or ad hoc basis, or as a distribution medium for the framework codebase for permanent installation elsewhere - for example, on non-Intel platforms or in non-Linux environments.

1.1.9.2 Capabilities

As a self-contained Linux distribution, the CD provides a complete environment for building and running a framework node, including the LAMP infrastructure. In addition, the framework codebase provides the required configuration information and scripts for setup of all node types, including database and web server setup.

1.1.9.3 Advantages

Ease of deployment

The distribution provides a complete, pre-configured environment, minimizing the user's learning curve and enabling rapid, productive use of the framework. We hope to simplify deployment of the framework to a series of point-and-click operations.



Minimal resource requirement

The distribution is designed to require the minimum investment in hardware and software resources. It is possible to set up an ad hoc cluster, using machines which are only temporarily available (e.g. machines ordinarily deployed as office systems, or machines whose regular users are temporarily absent): such systems can be booted from the CD into the framework environment and simply rebooted back into their normal roles when needed. At some point, the distribution will be equipped with zero configuration networking, enabling nodes to ``find'' each other and assemble into a cluster regardless of their physical location in an organization.

Also, as stated above, it is a property of the framework that it can be deployed on outdated or surplus equipment: the bootable distribution provides an ideal way to do this (for Intel hardware).

Finally, the distribution contains only Free/Open Source and public domain software: no additional licensing costs are incurred by organizations which choose to use it as shipped. At the same time, we endeavor to provide maximum functionality by including as complete a set of applications as possible (especially for unpacking archives).

1.1.9.4 Drawbacks

Lack of platform-specific packages

Certain proprietary and/or platform-specific packages cannot be included with the distribution, and we have yet to investigate the use of emulation to support the use of non-Linux applications. This is of particular importance to the processing of archive files. Nevertheless, where Free/Open Source or public domain alternatives exist, and we are aware of them, they are included.

Storage configuration

It may be necessary to deploy external storage, or to partition internal drives for cluster use, especially if the distribution is used to set up ad hoc clusters. This may require more expertise than is available.

Unfamiliarity with the operating environment

Lack of familiarity with the Linux environment may deter some users from using the distribution. We have tried to mitigate this effect by making setup and deployment of the framework as simple and intuitive as possible and by providing thorough documentation.

Performance penalty

As with any bootable CD distribution, when run from the CD, the performance of each node in a deployed cluster will suffer in comparison with a native installation. If machines are available, it is possible, and fairly straightforward, to permanently install the distribution to eliminate this problem.

Overall, we believe this distribution prototype holds much promise, and are currently testing it for use as the standard installation in the NSRL laboratory.

1.1.10 Conclusions

The NSRL framework is a public domain, platform neutral, modular and extensible system which enables the application of parallel processing to large data collections. Clusters built around the framework can be of arbitrary size, comprising systems of different architectures and operating systems, and may be dedicated or ad hoc in nature.



Cluster setup and operation is straightforward, being largely data driven, and relying primarily on the contents of a small number of configuration files. By the same token, the framework can be configured to carry out arbitrary operations on high-volume input data by simply flagging the target file type or types and slotting in the location of the desired application or script.

The framework is equally well suited for deployment in environments in which resource constraints dictate the use of older systems or temporary use of existing systems and in environments in which resources exist to deploy it on dedicated, high performance systems.

Finally, the framework is being integrated into a bootable CD distribution aimed at further facilitating its deployment, especially in resource-constrained environments.

1.1.11 Support

The National Software Reference Library program is a joint project of the National Institute of Justice (NIJ), the research and development organization of the U.S. Department of Justice; NIST's Office of Law Enforcement Standards (OLEs) and Information Technology Laboratory (ITL); and is supported by other organizations, including the Federal Bureau of Investigation, the Department of Defense Cyber Crime Center, and the Department of Homeland Security's Bureau of Immigration and Customs Enforcement and U.S. Secret Service.

1.1.12 References

[1] The National Software Reference Library homepage.

<http://www.nsrl.nist.gov/>

[2] Boland, T. and Fisher, G. (2000) ``Selection Of Hashing Algorithms''.

<http://www.nsrl.nist.gov/documents/hash-selection.pdf>

[3] KNOPPIX Linux Live CD

<http://www.knoppix.org/>

[4] Knoppix STD security tools distribution

<http://www.knoppix-std.org/>

[5] Helix Incident Response and Forensic tools distribution

<http://www.e-fense.com/helix/>

[6] Penguin Sleuth Bootable CD

<http://www.linux-forensics.com/index.html>

[7] Persistent Relational Objects in Perl (PROP).

<http://sourceforge.net/projects/prop/>

[8] The SNIPPETS C Source Code Archives.

<http://c.snippets.org/>

[9] White, Douglas and Tebbutt, John (2004) ``Digital Forensics - Using Perl to Harvest Hash Sets''. YAPC::NA::04, Buffalo NY. June 16-18, 2004.

<http://www.nsrl.nist.gov/documents/yapc2004/index.html>



1.1.13 Appendix 1: Database abstraction layer

1.1.13.1 Name

PROP - Persistent Relational Objects in Perl

1.1.13.2 Description

This module serves as a framework for both object and relationship persistence via a SQL capable database. Interaction with an underlying database is accomplished via Perl objects that abstract away all of the tedious construction, issuance, and processing of SQL queries. Within this framework, users may create, modify and delete objects, as well as link them together in arbitrary ways, and perform a variety of querying operations. The querying engine is reasonably sophisticated, allowing for a large degree of flexibility without becoming overly cumbersome, and performs intelligent buffering of results so as to be extensible to large queries.

Divided into several conceptual components, this module provides several well isolated sub-components that are reasonably easy to understand on their own. Short descriptions of these modules may be found in the following section.

1.1.13.3 SYNOPSIS

Create Database Tables

The first thing you need to do is to create the underlying database tables for object and relationship persistence. This is very simple. Object tables can have arbitrary fields, but must have a single automatically incrementing integer field as primary key. Link tables, for specifying relationships, must have a dual integer primary key, keys that will refer to the primary keys of rows in object tables, and then may also have arbitrarily many ``context'' fields that are used to specify information about the context from which an object was loaded.

Wrapping Database Functionality With Perl Classes

Once the database has been configured with the appropriate tables, the next thing to do is to write the Perl that will interface to it. For objects, the only thing required is to create classes that specify `PROP::Object` as the base class, and provide a `get_table_name` method that returns the name of the table in the database. For relationships, there is no subclassing involved; rather, one simply constructs objects of class `PROP::Link`, passing to its constructor the name of the link table in the underlying database, and the class names of the parent and child classes.

Create Application

Initialization of your application will entail working with the `PROP::Conf` class, in order to let the framework know where the database with which it will be working lives. Once the framework has been configured, working with objects, links, queries, and result sets is a seamless process that cleanly conceals the underlying database implementation. All of the classes know how to work with the database in a way that is (hopefully) totally transparent to the user. You just need to know how to employ their APIs to your end.

1.1.13.4 Modules

PROP::Object

If any class were said to lie at the center of this module, this would be the one. This class serves as the base class for all object types, and provides a collection of methods for performing all common object operations, such as saving, modifying, and deleting. Subclasses to this class must provide a `get_table_name()` method, and will probably also include various and sundry wrapper methods to flesh out the functionality of the class. One



can load single objects via themselves, or obtain collections of objects by using the `PROP::Query::Object` class.

PROP::Link

A `PROP::Link` object specifies a linking relationship in the database, and also provides mechanisms for creating, modifying and deleting relationships between objects as specified by the link in question.

PROP::Query::Object

This class is used to specify a query of a collection of objects. Queries of this type are specified with conditions, bindings of condition variables to values, and orderings. One may additionally specify a list of `PROP::Query::Link` objects which will specify that the objects loaded should be loaded with certain parent and/or children objects. An object of this class is passed to the constructor of `PROP::ResultSet::Object` to perform an actual query.

PROP::Query::Link

A `PROP::Query::Link` object is used to specify a query of a collection of linked objects. Queries of this type are specified with a link, conditions, and orderings. An object of this class is passed to the constructor of `PROP::ResultSet::Link` to perform an actual query.

PROP::ResultSet::Object

A `PROP::ResultSet::Object` is the source of objects as specified by a `PROP::Query::Object` object.. By repeatedly invoking the `get_next_result()` method until it returns `undef`, one can iterate over the results of a query, getting back instances of classes that derive from `PROP::Object`.

PROP::ResultSet::Link

A `PROP::ResultSet::Link` is the source of results of a query on a link as specified by a `PROP::Query::Link` object. By repeatedly invoking the `get_next_result()` method until it returns `undef`, one can iterate over the results of a query, getting back instances of the class `PROP::ResultSet::Link::Result`, a helper class for `PROP::ResultSet::Link`, the documentation for which may be found within the documentation for `PROP::ResultSet::Link`.

PROP::Schema

A `PROP::Schema` object contains all pertinent information about a database table. Users will probably never have need to create a `PROP::Schema` object directly. Rather, the `PROP::Object` class provides a `get_table()` method that returns an object of this type. The table returned is contingent upon the derived class's `get_table_name()` method. Using the name of the table, `PROP::Object` constructs a `PROP::Schema` object which populates itself with table structure information by querying the database.

PROP::SQL

The `PROP::SQL` class is a base class for several derived classes, each of which provides for a certain type of SQL statement. This class takes table names, field names, conditions, orderings, and other such statement parameters, and then via the `stringify()` method returns a scalar that holds the statement string. This tree of classes is mostly used internally by the framework to populate objects and manipulate the underlying database, but may also be useful to users who wish to issue queries that are not directly supported by the framework and thus need to craft custom queries.

PROP::Util



This class holds a collection of static utility methods that allow for performing common database tasks.

PROP::Conf

The PROP::Conf class is a base class for classes that parse connection configuration information for various RDBMS, e.g. PROP::Conf::MySQL.

PROP::DBH

The PROP::DBH class has but a single static method, `get_handle()`, which creates database handles via calling `DBI->connect(...)`. It pulls configuration information from PROP::Conf.