FLAIM

LOG ANONYMIZATION

FLAIM Core
Users Guide

LAIM Group
National Center for Supercomputing Applications
University of Illinois, Urbana-Champaign

February 28, 2008

Contents

Chapter 1

Overview and Prerequisites

FLAIM is a modular, open-source anonymization framework created for system/network/security administrators. **FLAIM** allows administrators to specify flexible, XML-based anonymization policies for their unique sharing needs. **FLAIM** is a versatile, multi-log/multi-level anonymization framework.

FLAIM has been developed to provide:

- A repository of up to the date and cutting-edge anonymization algorithms.
- Anonymization based on log type and log fields.
- A modular and flexible data input system that allows users to use **FLAIM** on their own, non-standard, logs.
- A policy manager that provides flexible anonymization schemes for each field of the log.

FLAIM is flexible because it can read many different logs. **FLAIM** has been built so that new logs can be anonymized very easily. This has been done by allowing developers to create I/O modules that parse proprietary log formats. The I/O modules will convert the proprietary format into **FLAIM**'s native format.

Because of this modular approach to **FLAIM** new logs can be added by anyone who chooses to create an I/O module. The creators of I/O modules are called ***FLAIM** Module Developers.*

FLAIM consists of two major components: **FLAIM**–Core and **FLAIM**–Modules. **FLAIM** core contains the anonymization library as well as a policy manager. **FLAIM**'s modules are input and output libraries for **FLAIM** . There are two sets of modules the basic modules and the extension modules. The basic modules were created by the LAIM group at NCSA and can be found with **FLAIM**–Core. These modules will be maintained compatible with future versions of **FLAIM**. The extension modules are developed by third-party users. For quality third-party modules that are being maintained, we will put links to

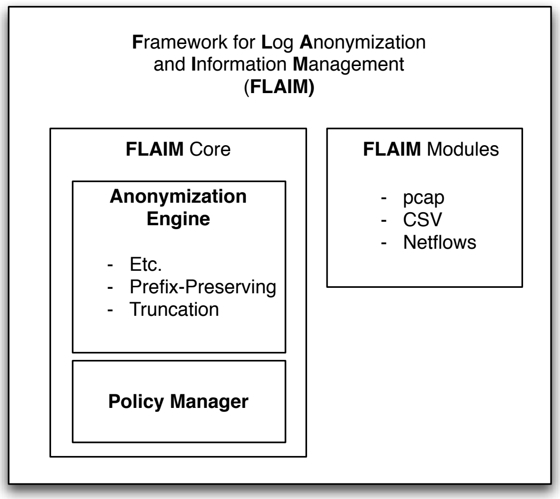


Figure 1.1: The architecture of **FLAIM**. **FLAIM**–Core is being developed by the LAIM group at NCSA. The basic **FLAIM** modules are also built by the LAIM group at NCSA, but these extension modules can be built by anyone by implementing the **FLAIM** API.

their download pages. However, the LAIM group maintains no responsibility for extension modules.

FLAIM–Core is developed by the LAIM Working Group at the NCSA. **FLAIM**-Core communicates to the modules via a module interface that every module must implement. This is described below in the section on modules. By separating parsing and I/O from the anonymization algorithms we allow third party users to create their own modules for their own datasets. This allows the flexibility of customizing **FLAIM** to your particular environment. We hope the third party consumers will make their modules available to the public, thus facilitating the sharing of logs between organizations.

The purpose of this document is to mostly describe how to use **FLAIM** for users. Details of **FLAIM**–Core and **FLAIM**–Module implementation are described as necessary, but abstracted away from the user as much as possible.

1.1 Prerequisites

While **FLAIM**–Core can be downloaded and installed without any modules, at least one module is needed to run **FLAIM**. Modules correspond to different types of logs, and you must download or created module for the type of log you wish to anonymize. A user must also specify a policy—an XML file specifying the manner in which to anonymize each field.

The **FLAIM** API used to communicate with the I/O modules is designed to be as general as possible. A module could be written for any data source which can be translated into a record and field format. Each record need not have the same subset of fields, but

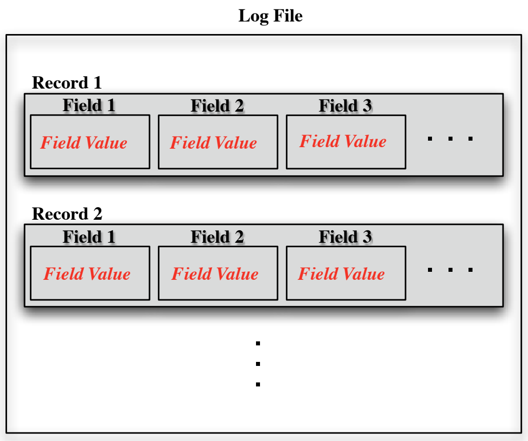


Figure 1.2: The abstract message exchange format for **FLAIM**.

the union of all the distinct field names must still be finite. Field names can be arbitrary strings. Figure 1.2 illustrates how a log is organized in the virtual format used between **FLAIM**–Core and **FLAIM**–Modules.

Anonymization algorithms with both *intra*-record and *inter*-record dependencies are implemented in **FLAIM**. All currently specified algorithms support streaming modes as complete random access is not necessary, just buffering. If an algorithm is later added that requires random access to the data source, there is a mechanism provided by which **FLAIM**–Core can communicate this to the I/O module currently loaded.

Chapter 2

Installation

As described in Section 1.1, **FLAIM** is composed of two parts: **FLAIM**–Core and **FLAIM**–Modules. To install and run **FLAIM** on your system, you must download and install both the **FLAIM**–Core installation package and at least one **FLAIM**–Module installation package. The installation packages can be found at: `http://flaim.ncsa.uiuc.edu/download.html`

As of version 0.5, **FLAIM** binaries are available as an RPM or Debian package for Linux. For FreeBSD, OpenBSD, NetBSD and Mac OS X, one must download and compile the source tarball. This method will work for any supported platform, and hence we describe the source installation procedure in this manual.

2.1 FLAIM–Core and FLAIM–Module dependencies

FLAIM–Core and **FLAIM**–Module depend upon a small set of libraries. These dependencies are listed below.

libxml2 Libxml2 is an open source C library for parsing and validating XML files. It is available on most Linux and BSD systems. It can be downloaded from `http://xmlsoft.org/`. Note that you may need both the binary library as well as the development files to compile **FLAIM**. For binary packages, you do not need the developers package.

libxslt Libxslt is a open source C library for parsing XSLT style-sheets. It is available on most Linux and BSD systems. It can be downloaded from `http://xmlsoft.org/XSLT/`. Note that you may need both the binary library as well as the development files to compile **FLAIM**. For binary packages, you do not need the developers package.

openssl Openssl is a open source C library for creating encrypted SSL tunnels which also provides many cryptographic functions. It is available on almost all Linux and BSD systems. It can be downloaded from `http://www.openssl.org/`. Note that you may need both the binary library as well as the development files to compile **FLAIM**. For binary packages, you do not need the developers package.

2.2 Installing FLAIM–Core source

To install **FLAIM**-Core, you must unpack the tarball, run the config script, and make the package:

```
[yoursystem]:$ tar zxf flaim-core-<version>.tgz
[yoursystem]:$ cd flaim-core-<version>
[yoursystem]:$ ./configure
.... lots of output here ......
[yoursystem]:$ make
[yoursystem]:$ make install
```

FLAIM modules are built in the same manner:

```
[yoursystem]:$ tar zxf flaim-module-<modulename>-<version>.tgz
[yoursystem]:$ cd flaim-module-<modulename>-<version>
[yoursystem]:$ ./configure
.... lots of output here ......
[yoursystem]:$ make
[yoursystem]:$ make install
```

2.2.1 Important Note on RPMs and Debian packages

We compile binaries on Debian 3.1 (¡i¿Sarge¡/i¿) and use the ¡i¿alien¡/i¿ package to generate RPMs and Debian packages from binary tarballs. As a consequence, you may have an issue using an RPM on a very old or very new Linux distribution because of differences in the major versions of critical libraries. For example, Fedora Core 6 (but not Core 4) uses ¡i¿libstdc++¡/i¿ version 6 as opposed to version 5 in Debian Sarge, and hence it will not install our RPM because of a dependency issue. In this case, and similar situations, one can simple download the source and compile FLAIM just fine.

2.2.2 Installing in Non-Standard Locations

The default installation behavior is to create */usr/local/flaim* and place all the libraries and configuration files there. In addition, a symbolic link is made to */usr/bin/flaim.* One must normally be root to install in these directories, or to install the man page.

To install **FLAIM** in a different location, one can pass an option to the *configure* script. The option to set is *prefix.* The default value for this variable is */usr/local.*

FLAIM then installs to *$(prefix)/flaim*, i.e. */usr/local/flaim*. Say, instead, a user wants to install **FLAIM** into */usr/flaim*. He would then pass the option to the *configure* script as follows:

```
[yoursystem]:$$ ./configure --prefix=/usr
```

If the installation directory is changed for **FLAIM**-Core, it **must** be changed in the same way when modules are installed. Again, this can be done with the same options for their *configure* scripts. Note that there is no way to install in non-standard locations when using binary packages like RPMs or Debian packages.

One can also install the prerequisite libraries in non-standard locations. To do this, they should pass the *with-lib-prefix* option to the *configure* script. This can be done as follows, where *lib-location* is a variable indicating the directory containing the library shared objects.

```
[yoursystem]:$$ ./configure --with-lib-prefix=$lib-location
```

2.3 FLAIM Directory structure

The **FLAIM** directory is organized like this:

```
FLAIM_ROOT
|
| - modules
|
| - config
```

The modules directory contains the files (organized as modules) necessary to read and write different log files. The config directory contains the various xml files necessary to validate the anonymization policy. Details of the organization of these directories are described in the *Developer's Guide*.

Chapter 3

How to use FLAIM

Before one can use **FLAIM** to anonymize a log they must have three things:

1. an installed version of **FLAIM**–Core
2. a **FLAIM** module for the type of log they wish to anonymize, and
3. an anonymization policy.

Instructions for installing **FLAIM**–Core and **FLAIM** modules are found in Chapter 2. Details on writing an anonymization policy are covered in Chapter 4.

3.1 Example Usage

Suppose the module name is "netflow", and that the data anonymization policy is specified in "netflow-anony1.xml". The simplest command to run **FLAIM** would be:

```
flaim -m neflow -p netflow-anony1.xml -i input.file -o output.file
```

If modules are not instatlled in the default location (e.g., a user without root access has installed additional modules), one must use the **-M** and **-s** flags to indicate where the module library and module schema, respectively, are located.

For instance, suppose the library is in ~/lib/libpcap.so and the module schema is in ~/m-pcap-schema.xml. The one could invoke **FLAIM** to use this module with:

```
flaim -M ~/lib/libpcap.so -s ~/m-pcap-schema.xml -p pcap-anony1.xml \
-i input.file -o output.file
```

3.2 Command Line Options

-h –help Display the usage information and exit.

-i –input <input file> Specifies the source log for anonymization. If unspecified, stdin will be used. However, not all modules support reading input from stdin. Those that do not support streaming will exit and force you to specify a file name.

-l –list Lists all installed modules. **FLAIM** will not find manually installed modules in non-standard locations.

-m –module <module-name> Load the specified parsing module. The **-l** option shows the valid choices. Either this option must be used to specify a module installed in the default location, or the **-M** option must be used.

-M –moduleLib <module-lib-path> Load the module library from the given path. This option is mutually exclusive with **-m** and used for explicitly specifying the module location. It is also necessary to specify the schema location using **-s** when using this option.

-o –output <file-name> Specifies the destination file for anonymized data. If unspecified, stdout will be used. However, not all modules support writing output to stdout. Those that do not support streaming, will exit and force you to specify a file name.

-p –policy <file-name> The use of this flag is mandatory as it specifies the location of the user policy.

-s –schemaModule <module-schema-file-path> Load the module schema from the file specified with this option. This option is used if and only if the **-M** option is used.

-v –verbose Print verbose messages to stderr.

-V –version Print version information to stderr and exit.

-x –xtraConfig <file-name> This is used to specify a file containing extra information to be passed to the parsing module. It is optional and ignored by most modules.

3.3 Error Handling

FLAIM outputs error messages to *STDERR* and informational messages to *STDOUT* (Or course, we cannot enforce this behavior in third party modules). Sometimes a module will either be unable to parse a packet properly, or it will come across a corrupted record. Every effort is made to continue, but sometimes records are corrupted badly enough that re-synchronization is impossible. In this case, **FLAIM** must exit without completing the task.

If the corruption isn't bad or **FLAIM** just isn't able to fully parse the record (e.g., because it has come across an obscure IP protocol), **FLAIM** will continue processing records while writing the troublesome records to an error file. The naming convention for this file is lib*module_name*flaim.errors. This file will contain a subset of the records from the original log, namely, the troublesome records. If the records are not too corrupted, one should be able to process this error file with the same tools they would process the original log. For example, the error file created by the pcap module should be something that tcpdump can process.

Chapter 4

Writing FLAIM Policies

Anonymization policies allow users to specify how to anonymize a log. A policy identifies which anonymization algorithm should be used for each field in a log. As mentioned above, we consider a file to be a series of records, and each record to consist of a series of fields. Each field is represented by the field name and a pointer to the value of the field. An anonymization policy is a listing of field names and anonymization algorithms with their respective options. In this document, the term "policy" always refers to an anonymization policy.

An anonymization policy must be *well-formed* and *valid*. To be well-formed a policy must be written in a certain structure. We check that a policy is well-formed by validating it against the ***FLAIM*** *Schema*.

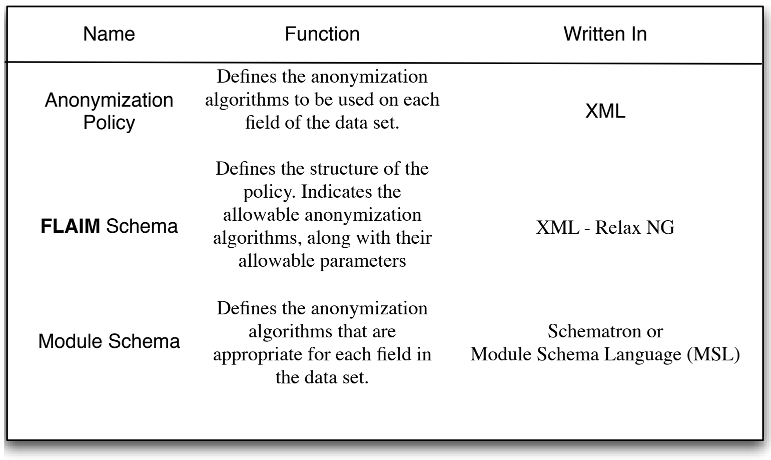
A policy is valid if it specifies fields that exist in the data and anonymization algorithms that exist—with correctly specified options. In addition, a policy must specify anonymization algorithms that are applicable to the type of data in the field (.e.g, anonymization algorithms for strings cannot be applied to binary fields). A policy is considered valid if it is successfully validated against the ***FLAIM*** *Schema*—which specifies the existing anonymization algorithms an options—and the *Module Schema* which specifies the valid field names and anonymization algorithm combinations.

Table 4.1 summarizes the functions of the polices.

4.1 Anonymization Policy

The anonymization policy, starts with a ⟨*policy*⟩ tag. This indicates the beginning of a policy. After the policy tag, there are one or more ⟨*field*⟩ tags. Appendix A contains reference information about all the tags used in **FLAIM**.

The policy tag indicates that this file is an anonymization policy. Each field tag indicates the name of a field, and the anonymization algorithm to be used on that field. The field tag takes an attribute called *name*. The value of the *name* attribute is the name of



Name	Function	Written In
Anonymization Policy	Defines the anonymization algorithms to be used on each field of the data set.	XML
FLAIM Schema	Defines the structure of the policy. Indicates the allowable anonymization algorithms, along with their allowable parameters	XML - Relax NG
Module Schema	Defines the anonymization algorithms that are appropriate for each field in the data set.	Schematron or Module Schema Language (MSL)

Figure 4.1: Summary of the policies and schemas used in FLAIM

the field to be anonymized. The module user guides will indicate the valid field names for each type of log.

After specifying a field name, a single anonymization algorithm must be specified. There can only be one anonymization algorithm per field. The full list of anonymization algorithms tags are given in Appendix A, and a summary of the anonymization algorithms themselves are given in Appendix B

The 6 line example below shows a very simple user policy. Line 1 indicates that the file is an anonymization policy. Lines 3–5 specify the anonymization algorithm with options for the field named *time*. Line 3 specifies that the *time* field should be anonymized by the Time Unit Annihilation anonymization algorithm, and line 4 indicates that the time unit to be annihilated is the “seconds” part.

```
1 <policy>
2   <field name="time">
3     <TimeUnitAnnihilation>
4       <timeField>Seconds</timeField>
5     </TimeUnitAnnihilation>
6 </policy>
```

4.2 Module Schema

Not all anonymization algorithms can operate on every type of field. For instance, it does not make sense to apply the prefix-preserving anonymization algorithm for IP addresses on usernames. Since only Module Developers are aware of the type and meaning of the log fields, they are the ones that specify what anonymization algorithms make sense for each field in the data set. As developers of **FLAIM**–Core we cannot anticipate every possible field type yet alone the field names a module developer may use.

The module schema can be written as a Schematron file, or in a simple language we have dubbed "Module Schema Language". Detailed knowledge of the module schema is not necessary for using **FLAIM**. Users should consult the modules users guides to know what anonymization algorithms can be used for which fields.

For more information on writing a module schema, consult the **FLAIM**–Module Developer's guide.

4.3 FLAIM Schema

The **FLAIM** schema specifies allowable anonymization algorithms along with the valid options for each algorithm. It is unnecessary for users or module developers to modify this schema. However, users must know what anonymization tags and options are available. This is covered in Appendix A in detail.

4.4 An example

To make policy creation more concrete, let us look at a slightly more realistic example. We will use a simple variant of NetFlows as our data source. NetFlows are a higher level abstraction than pcap with each record essentially representing a socket. Our simplified NetFlow logs will have 5 fields:

SrcIP The IP address of the source host

DstIP The IP address of the destination host

StartTime Starting time of the flow

EndTime Ending time of the flow.

Size The number of packets transferred for the duration of the flow.

Each record will represent 1 flow. We will use the very small log below for our example (note that the no. column is not part of the log; it is just there for reference purposes):

No.	SrcIP	DstIP	StartTime	EndTime	Size
1	141.142.3.5	132.245.8.9	90123	90125	789
2	132.267.2.6	141.167.2.9	12345	12380	1000
3	345.67.234.12	132.267.2.6	5678	5680	12

Assume the module developer has created a module to read and write log files of this type. They have also specified a module schema that identifies the anonymization algorithms available for each field which we illustrate below.

```
<?xml version="1.0"?>
<modulepolicy>
  <constraint>
    <fieldname>SrcIP</fieldname>
    <allowedAnony>
      <Truncation/>
      <Random_Permutation/>
      <Prefix-Preserving/>
    </allowedAnony>
  </constraint>

  <constraint>
    <fieldname>DstIP</fieldname>
    <allowedAnony>
      <Truncation/>
      <Random_Permutation/>
      <Prefix-Preserving/>
    </allowedAnony>
  </constraint>

  <constraint>
    <fieldname>StartTime</fieldname>
    <allowedAnony>
      <TimeUnitAnnihilation/>
      <RandomShift/>
      <Enumeration/>
    </allowedAnony>
  </constraint>

  <constraint>
    <fieldname>EndTime</fieldname>
    <allowedAnony>
      <TimeUnitAnnihilation/>
      <RandomShift/>
      <Enumeration/>
    </allowedAnony>
  </constraint>

  <constraint>
    <fieldname>Size</fieldname>
    <allowedAnony>
      <Truncation/>
      <BlackMarker/>
      <MultiLateralClassification/>
    </allowedAnony>
  </constraint>
</modulepolicy>
```

The functionality of these anonymization algorithms are described in Appendix B. The module schema should be available with the module in XML form, or as part of the module documentation.

Given this module schema, the user can reference Appendix A to find the appropriate options for each of these anonymization algorithms. To construct an anonymization policy, she would create an xml file that has one of the anonymization tags per field name specified—though, not every field need be specified in the policy. Below, we create a sample policy using the field names specified in the example module schema.

First, the user must pick the fields to be anonymized. We will begin with `SrcIP`. There are three valid anonymization algorithms for this field specified in lines 5–9 of the module schema. We will choose the `<IPv4Prefix-Preserving>` algorithm. Referencing the Appendix B, we see that the entry for `<IPv4Prefix-Preserving>` is this:

<*BinaryPrefixPreserving*> . . . </*BinaryPrefixPreserving*>

Function: Anonymization Algorithm. Prefix preserving anonymization maintains the constraint that the first n bits of the unanonymized data match, if and only if, the first n bits of the anonymized data match. In our implementation a passphrase is required as a key to the anonymization. Thus, it is a deterministic algorithm unlike some implementations.
This algorithm could be used on any data type between 1 and 16 bytes, and thus can work on IPv4, IPv6 and MAC addresses. **FLAIM**–Core simply expects an array of unsigned chars ordered from most to least significant bytes. Thus, if a module passes something like a UINT32, it must make sure it is in *network byte order* first because **FLAIM**–Core treats it as an unsigned char array in order to handle larger sized data types like IPv6 addresses. Therefore, this algorithm can be sensitive to endian issues, and the module developer must take care of them.

Parameters: ***<passphrase> ... </passphrase>***

Function Provides a key to the anonymization algorithm.
Usage: Required
Allowed Values: Any

The function section describes what kind of tag this is, (*Anonymization Algorithm* in this case), and what it does. The attributes section lists any attributes this tag takes. The Parameters sections lists any sub-tags that can be written within the anonymization algorithm's tag.

Each parameter has three pieces of information associated with it. The first, *Usage*, indicates whether this parameter is required or optional. In this case the `<Passphrase>` parameter is required. The *Allowed Values* line indicates what values can be specified in this parameter. Finally, the *Function* line indicates what this parameter actually does to affect the algorithm.

The simple anonymization policy looks like this now:

```
<policy>
    <field name="SrcIP">
        <IPv4Prefix-Preserving>
          <Passphrase>abracadabra</Passphrase>
        </IPv4Prefix-Preserving>
    </field>
```

Figure 4.2: Simple anonymization policy

The user continue in this manner and pick other anonymization algorithms for each of the fields. Here is one possible finished policy:

```
<policy>
    <field name="SrcIP">
        <Prefix-Preserving>
          <Passphrase>abracadabra</Passphrase>
        </Prefix-Preserving>
    </field>

    <field name="DstIP">
        <Truncation>
          <NumToTruncate>8</NumToTruncate>
        </Truncation>
    </field>

    <field name="StartTime">
        <TimeAnnihilation>
          <PortionToAnnihilate>Year</PortionToAnnihilate>
        </TimeAnnihilation>
    </field>

    <field name="EndTime">
        <TimeAnnihilation>
          <PortionToAnnihilate>Month</PortionToAnnihilate>
        </TimeAnnihilation>
    </field>

        <field name="Size">
        <BlackMarker>
          <NumToTruncate>All</NumToTruncate>
          <MarkerValue>0</MarkerValue>
        </BlackMarker>
    </field>


```

Figure 4.3: Anonymization Policy

Appendix A

Tags in the User Policy

<*BinaryPrefixPreserving*> . . . </*BinaryPrefixPreserving*>

FUNCTION: Anonymization Algorithm. Prefix preserving anonymization maintains the constraint that the first n bits of the unanonymized data match, if and only if, the first n bits of the anonymized data match. In our implementation a passphrase is required as a key to the anonymization. Thus, it is a deterministic algorithm unlike some implementations.
This algorithm could be used on any data type between 1 and 16 bytes, and thus can work on IPv4, IPv6 and MAC addresses. **FLAIM**–Core simply expects an array of unsigned chars ordered from most to least significant bytes. Thus, if a module passes something like a UINT32, it must make sure it is in *network byte order* first because **FLAIM**–Core treats it as an unsigned char array in order to handle larger sized data types like IPv6 addresses. Therefore, this algorithm can be sensitive to endian issues, and the module developer must take care of them.

PARAMETERS:

<***passphrase***> ***...*** </***passphrase***>

Function Provides a key to the anonymization algorithm.
Usage: Required
Allowed Values: Any

<*Hash*> . . . </*Hash*>

Function:

This anonymization algorithm implements a cryptographic hash [or HMAC] on a given field. If the field is string data, then the hash output is represented as a string. Specifically, it is the hexadecimal representation of the binary hash value printed as a string. If the field is binary, then the output remains binary.

One problem with binary fields is that the anonymized value should be the same size as the unanonymized value (as not to break log parsers), but the hash output is always the same size for a given hash algorithm. Therefore, if the hash output is too long, we truncate it to be the same size as the input. Intuitively, this seems to weaken the hash, but the hash applied to a small input space can never be stronger than a brute-force dictionary attack on the small input space anyway. For example, a hash on the 32 bit IPv4 address space is never going to be terribly strong because the space is small enough to brute force whether we use MD5 or SHA-512. The only way to make that less feasible would be to create a very slow hash algorithm, perhaps by hashing the hash several times. This would still only make a linear difference with respect to the input space size.

If the input space is larger than the output of the hash, then we simply pad the output to make it the same length. Therefore, we do not break log parsers expecting a field to be a certain size. This neither weakens, nor strengthens the hash. The method we use for padding is just to pad with 0's for the empty least significant bits.

This function can also be configured as an HMAC (Hash Message Authentication Code). If a password is specified, it becomes a key to the HMAC. If unspecified, the function behaves as just a normal hash.

This anonymization algorithm is the reason that we rely on openssl being installed as a prerequisite. Some of the hash algorithms may be unavailable if you are running an older version of openssl.

Parameters:

<passphrase> ... </passphrase>

Function Behaves as an HMAC with the passphrase acting as a key.
Usage: Optional
Allowed Values: Any string.

<digestAlgorithm> ... </digestAlgorithm>

Function Specifies the hash algorithm to use.
Usage: Optional (Default MD5)
Allowed Values: MD2, MD5, SHA, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, RIPEMD-160 (Availability dependent upon openssl version.)

<*StringTruncation*> ... </*StringTruncation*>

Function:

Anonymization Algorithm. StringTruncation removes 8-bit characters from a string. Direction can be chosen.

Parameters:

<numChars> ... </numChars>

Function The number of characters to truncate.
Usage: Required
Allowed Values: Integer — any value.

<direction> ... </direction>

Function Indicates the direction to shift
Usage: Optional (default is from the right)
Allowed Values: String – “left” or “right”

<*NumericTruncation*> ... </*NumericTruncation*>

FUNCTION: Anonymization Algorithm. Numeric Truncation right-shifts a numeric field. The amount of shift depends on the NumShifts parameter and the radix parameter. Radix indicates the base of the numeric value. So if radix is 2, and NumShifts is 3, the value is shifted to the right by 3 bits. If the radix is 10, and NumShifts is 3, then the value is right shifted by 3 decimal places. We do shifting for binary values since the number of bits in a binary field is typically fixed.

PARAMETERS:

<numShifts> ... </numShifts>

Function Describes the number of bits to truncate.
Usage: Required
Allowed Values: Integer — any value.

<radix> ... </radix>

Function Indicates the base of the value.
Usage: Required
Allowed Values: Integer — 2 or 10.

<direction> ... </direction>

Function Indicates the direction to shift
Usage: Optional (default is from the right)
Allowed Values: String – “left” or “right”

<BinaryTruncation> . . . ***</BinaryTruncation>***

FUNCTION: Anonymization Algorithm. BinaryTruncation removes a specified number of bits from a byte array. Direction can be specified.

PARAMETERS:

<numBits> ... </numBits>

Function The number of bits to truncate.
Usage: Required
Allowed Values: Integer — any value.

<direction> ... </direction>

Function Indicates the direction to shift
Usage: Optional (default is from the right)
Allowed Values: String – “left” or “right”

<*BinaryBlackMarker*> ... </*BinaryBlackMarker*>

FUNCTION:

Anonymization Algorithm. Binary Black Marker replaces the n least significant bits of a binary field with either 0’s or 1’s.

PARAMETERS:

<numMarks> ... </numMarks>

Function Specifies the number of least significant bits to “Black Out”.
Usage: Required
Allowed Values: Any non-negative integer.

<replacement> ... </replacement>

Function Sets the replacement value for the n least significant bits of the binary data.
Usage: Required
Allowed Values: Either 0 or 1.

<*StringBlackMarker*> ... </*StringBlackMarker*>

FUNCTION:

Anonymization Algorithm. String Black Marker replaces the last n characters of a string with the character c.

PARAMETERS:

<numMarks> ... </numMarks>

Function Specifies the number of characters to "Black Out" from the end of the string.
Usage: Required
Allowed Values: Any non-negative integer.

<replacement> ... </replacement>

Function Sets the value that will replace the last n characters of the string. This value will be repeated n times.
Usage: Required
Allowed Values: Any single character.

<TimeUnitAnnihilation> ... *</TimeUnitAnnihilation>*

FUNCTION:

Anonymization Algorithm. Annihilates a specific part of the time field, either the Year, Month, Day, Hour, Minute or Second portion. Time must be represented as number of second from epoch.

PARAMETERS:

<timeField> ... </timeField>

Function Describes the portion of the time stamp to annihilate
Usage: Required
Allowed Values: String — either "years", "months", "days", "hours","minutes", "seconds"

<secondaryField> ... </secondaryField>

Function: Indicate another field that will be annihilated in the same way.
Usage: Optional
Allowed Values: String — Name of another field.

<RandomTimeShift> ... *</RandomTimeShift>*

FUNCTION:

Anonymization Algorithm. Shifts timestamps by random value. The amount of shift will be at least lowerTimeShiftLimit, but not greater than upperTimeShiftLimit. The unit by which time is shifted is in seconds.

PARAMETERS:

<lowerTimeShiftLimit> ... </lowerTimeShiftLimit>

Function: Specifies the minimum amount of shift.
Usage: Required
Allowed Values: Numeric — time_t Unsigned 32 bit integer

<upperTimeShiftLimit> ... </upperTimeShiftLimit>

Function: Specifies the maximum amount of shift.
Usage: Required
Allowed Values: Numeric — time_t Unsigned 32 bit integer

<secondaryField> ... </secondaryField>

Function: Indicates another field in the record which will be shifted the exact same amount as this field.
Usage: Optional
Allowed Values: String — field name.

<TimeEnumeration> ... *</TimeEnumeration>*

FUNCTION:

Anonymization Algorithm. Preserves order, but not absolute time or time between records. A random time is chosen for the first record. Each subsequent record is a standard time unit later than the previous if the time stamps are not equal. Since the records might not be linearly increasing in time a buffer is used to sort the records. This buffer size can be specified. The standard time unit between records can also be set.

PARAMETERS:

<intervalSize> ... </intervalSize>

Function: Number of seconds between records.
Usage: Required
Allowed Values: Integer — Number of seconds. uint 32 value.

<bufferSize> ... </bufferSize>

Function Size of the lookahead buffer.
Usage: Optional
Allowed Values: Integer: Any non-negative integer. uint32 value

<secondaryField> ... </secondaryField>

Function Indicates another timestamp field which will be anonymized in the same way as this field.
Usage: Optional
Allowed Values: String — Any field name.

<baseTime> ... </baseTime>

Function Time stamp of first record.
Usage: Required
Allowed Values: Integer: Any non-negative integer. uint32 value.

<TimeRandomPermutation> . . . *</TimeRandomPermutation>*

FUNCTION: Anonymization Algorithm. Substitutes a random value for the time field.

ATTRIBUTES: None.

PARAMETERS:

<secondaryField> ... </secondaryField>

Function: Should be set to another time field. If so, the difference will be preserved between the fields after anonymization.
Usage: Optional
Allowed Values: String — Name of another field.

<***Classify***> . . . </***Classify***>

FUNCTION:

Anonymization Algorithm. Takes a numeric field and bins its values. For example, bilateral classification of port numbers can separate ephemeral ports from non-ephemeral ports by bining port numbers below 1024 in one bin, and ones greater than or equal to 1024 in another. The user must specify the constraints for the bins.

PARAMETERS:

<***configString***> ***...*** </***configString***>

Function Bins for Multi-Lateral Classification

Usage: Required

Allowed Values: A string of monotonically increasing comma separated numeric values, i.e.: 2.1, 3.4, 4.3. A string of the form "a:x,b:y,c:z" will result in all values less then a to be binned to x; all values less than b but greater than a to be binned in y; and so on. If the value is greater than c, no binning will be performed.

<***BinaryRandomPermutation***> . . . </***BinaryRandomPermutation***>

FUNCTION:

Anonymization Algorithm. Randomly permutes the values of a field via the use of random hash tables. The random mapping is determined both by the data set and the state of the PRNG. Therefore, it is not deterministic and will yield different results on different runs. *A future implementation may be made more memory efficient through use of Bloom filters.*

PARAMETERS:

<***HostBlackMarker***> . . . </***HostBlackMarker***>

FUNCTION:

Anonymization Algorithm. Anonymizes a fully qualified host name.

Parameters:

<Type> ... </Type>

Function Indicates which of the parts of the hostname to anonymize. FullName replaces the entire hostname/domain name. HostOnly replaces only the host.

Usage: Required

Allowed Values: String — either "FullName", or "HostOnly"

<hostReplacement> ... </hostReplacement>

Function Value to replace the host part of the hostname

Usage: Required

Allowed Values: String

<domainReplacement> ... </domainReplacement>

Function Value to replace the domain part of the hostname

Usage: Required

Allowed Values: String

<HostHash> . . . </HostHash>

Function:

Anonymization Algorithm. Creates a hash of the hostname.

Parameters:

<type> ... </type>

Function Indicates which type of hashing to do.

Usage: Required

Allowed Values: String — only option is "MD5"

<Annihilation> . . . </Annihilation>

Function:

Anonymization Algorithm. Replaces numeric field with 0, replaces string field with character 0

Parameters:

None.

<***Substitution***> . . . </***Substitution***>

FUNCTION: Anonymization Algorithm. Substitutes field with value specified.

PARAMETERS:

<***substitute***> **...** </***substitute***>

Function Substitution value
Usage: Required
Allowed Values: Value to substitute

Appendix B

Anonymization Primitives

FLAIM implements many *types* of anonymization algorithms which we call *anonymization primitives*. While specific algorithms—listed in A—are often tightly coupled with the data type being anonymized, these primitives are usually not. For example, we have 3 algorithms to do what we call *truncation*: one for binary data, one for strings, and one for numeric values of any base. The options and semantics of truncation change slightly for each type of data. In this appendix, we present these different anonymization primitives in as much generality as possible.

B.1 Black Marker

Black marker anonymization is equivalent, from an information theoretic point of view, to printing out a log and going over each value of a sensitive field with a black marker. This analog variant is often seen in sensitive documents retrieved from the government. We simply implement a digital equivalent.

In our implementation, the entire field or just part of it can be "blacked out". So IP addresses could have just the last octet "blacked out", (e.g., 192.168.77.99 could map to 192.168.77.0). In this case, we have replaced the last 8 bits of the IP address with 0's (Though some would call this truncation). With the *BinaryBlackMarker* algorithm used here, we could have substituted with 0 bits or 1 bits. Some instances of black marker anonymization allow us to substitute with different values of a coarser granularity than a single bit. For example, using *HostBlackMarker* we can anonymize the hostnames by replacing every hostname in a log with *host@example.net*.

B.2 Truncation

Truncation works differently for binary and string data. For a string value, you choose some middle point—not necessarily defined as a fixed number of characters from the beginning—

and cut the string off after that point. For example, one could truncate the domain information from an e-mail address so that “user@example.net” is replaced with just “user”.

Truncation will shorten strings, but this is a problem for fixed length binary values. You could simply replace all trailing values with 0’s, but we have called this black marker anonymization. Keeping with the idea that truncating is shortening a value, we pick a point for truncation and right shift until all bits to the right of this point are shifted off the end. For example, consider IP addresses represented as a binary 32 bit unsigned integer. The IP address 192.168.77.88 could have the last 16 bits truncated which would result in 0.0.192.168.

Sometimes we do not want to work on a number as binary when truncating. Take the decimal number 11977. We could truncate the last two digits, instead of bits, and come up with 119. Phone numbers, social security numbers and zip codes are all examples of data where it makes sense to use base 10 truncation instead of binary shifts.

Note that truncation and black marker still are not 100 percent mutually exclusive. For example, truncating all 32 bits and replacing all IP’s with the constant address of 0.0.0.0 will have the same result.

B.3 Permutation

In the most general sense, a *permutation* is a one-to-one and onto mapping on a set. Thus, even a block cipher is a type of permutation. There are many ways that permutations can be used. For larger binary fields, it usually makes sense to use a strong, cryptographic block cipher. Thus, if one wishes to use the same mapping later, they just save the key. This is excellent for fields like the 128 bit IPv6 address. However, there are no strong 32 bit block ciphers to do the same for IPv4 addresses. Using a larger block would require padding, and the output of the anonymization function would be larger than the input. In these cases, one can use tables to create random permutations. The problem is, of course, that one cannot save these tables as easily as a cryptographic key to keep mappings consistent between logs anonymized at different times.

In addition to random permutations that are cryptographically strong, there is sometimes a need to use structure preserving permutations for certain types of data. When differences between values must be preserved—often the case with timestamps—a simple shift can be used. Shifting all values by a certain number may be an acceptable way to permute values in some instances. For IP addresses, the subnet structure may need to be preserved without knowing the actual subnets (e.g., when analyzing routing tables). Prefix-preserving pseudonymization is a type of permutation that preserves exactly this type of structure. **FLAIM** implements several types of permutations for different types of data.

B.4 Hash

Cryptographic *hash* functions can be useful for anonymization of both text and binary data. The problem with binary data, of course, is that one must often truncate the result of a hash function to the shorter length of the field—**FLAIM** is careful not to increase the byte size of binary data in its output as that could break other parsers that operate on the data. This also means that the hash function is weaker and has more collisions. For fields 32 bits or smaller, dictionary attacks on hashes become very practical. For example, it is well with the capability of a modern adversary to create a table of hashes for every possible IPv4 address. The space of possible values being hashed is too small. Thus, hash functions must be used carefully and only when the possibility of collision in the mappings is acceptable. Often, it is better to use a random permutation. For string values, **FLAIM** outputs the hash in an ASCII representation of the hexadecimal value.

B.5 HMAC

HMACs (Hash Message Authentication Codes) are essentially hashes seeded with some secret data. Adversaries cannot compute the HMAC themselves without access to the key, and thus they cannot perform the dictionary attacks that they could on simple hashes. In cases where hashes are vulnerable to such attacks, HMACs make sense. However, HMACs, like hashes, are inappropriate where collisions are unacceptable.

B.6 Partitioning

Partitioning is just what it sounds like. The set of possible values is partitioned into subsets—possibly by a well-defined equivalence relation—and a canonical example for each subset is chosen. Then, the anonymization function replaces every value with the canonical value from the subset to which it belongs. Black marker anonymization and truncation are really just special cases of partitioning. For example, say that the last 8 bits of IPv4 addresses are “blacked out” with 0’s. Then the set of IP addresses is being partitioned into class C networks. Furthermore, the canonical representation is simply the network address of the class C subnet. However, partitioning is not always so simplistic, and our next type of anonymization algorithm is a very unique type of partitioning for timestamps.

B.7 Time Unit Annihilation

Time unit annihilation is a special type of partitioning for time and date information. Timestamps can be broken down into year, month, day, hour, minute and second subfields. When this is done, one can annihilate any subset of these time units by replacing them with 0. For instance, if she annihilates the hour, minute and second information, the time has been removed but the date information retained—actually, a type of black marker. If she wipes out the year, month and day, the date information is removed but the time is unaffected. It is clear that this is a very general type of partitioning, but it still cannot partition in arbitrary ways. For instance, it cannot break time up into 10 minute units.

B.8 Enumeration

Enumeration can be very general, though **FLAIM** currently uses it as just an option for timestamps. However, enumeration would work on any well-ordered set. Enumeration, will first sort the records based on this field, choose a value for a first record, and for each successive record, it will choose a greater value. This preserves the order but removes any specific information. When applied to timestamps, it preserves the sequence of events, but it removes information about when they started or how far apart two events are temporally.

A straightforward implementation could sort, choose a random starting time, and space all distinct timestamps apart by 1 second. Note that the output doesn't actually have to be reordered records. For example, the set $\{1.2, 5.6, 7, 0.3, 9.3, 4.8\}$ could be enumerated with the elements left in place as follows $\{2, 4, 5, 1, 6, 3\}$. However, sorting on the anonymized and unanonymized field values would produce the same result as order is preserved by enumeration. ***FLAIM** actually approximates sorting using a greedy algorithm that runs in real time. This is done for both efficiency and so that we may work on streamed data. **FLAIM** can be configured to be more or less aggressive in its sorting depending on how disordered one expects the data to be.*

Appendix C

Copyright

C.1 Copyright