

# **Signing and Verifying SNOOPY-files**

**by Matthias Dube**

**March 2005**

# Table of Contents

1. Task
2. Motivation
3. Fingerprinting and MD5
4. Implementation and Integration into SNOOPY
  - 4.1 Signing a SNOOPY-file
  - 4.2 Verifying a SNOOPY-file
5. The SIVE-Tool
6. Implementation Details and Code Samples
7. Tests and Performance
  - 7.1 Testing
  - 7.2 Performance checking
8. References

## **1. Task**

The task was to develop a simple but sufficient method to sign and verify graphs which have been created with the net editor SNOOPY. The implementation of this feature should then be integrated into the editor. Furthermore a command line tool should also be developed in order to have the opportunity to use the same functionality apart from SNOOPY.

## **2. Motivation**

Petri nets are becoming more and more popular, especially for applications with high dependability demands, in particular speaking of safety and/or security. Over the years many analysis techniques, which work on the Petri net model, have been developed in order to predict a systems possible behavior and they have been implemented in countless tools.

However, to get a useful analysis of a system, it is essential to create a reliable model, which exactly meets the required properties and specifications.

Therefore it is not uncommon that people spend many hours on developing such a model including its validation and verification. Unfortunately many real world application models tend to become very big very fast. For that reason and because of the complexity of some analysis algorithms, the time for analyzing these models may easily exceed several days. In addition the resulting protocols have to be checked by human intelligence which in fact is a very expensive resource. In other terms the procedure of creating a reliable model of a specified system with certain properties is a long, resource consuming process.

Looking at this there is a strong demand in reliability of both, the model itself and also its verification/validation which is usually granted by some kind of certification stamp given as a result of the developing process.

Nevertheless throughout its life time a net model may encounter many occasions where it might be changed. There are basically two reasons for that.

One is a mistakenly done destruction. That includes many possible causes such as hardware faults due to wear out or external influences, software faults and user faults.

The other one is the destruction on purpose. Particularly models that control sensible equipment, like reactive or embedded systems, may be target of criminal attacks which end up in faulty behavior.

Not depending on the actual cause, whether by mistake or on purpose, the result of such destruction could be misleading or even disastrous.

As already mentioned Petri net models are likely to become very complex, leading to use of a hierarchical structure. Although this practice of distributing the net into different levels of abstraction helps to understand the basic functionality, it also makes it very difficult to find changes in the net that have been made and nearly impossible to find them at first sight.

One way to oppose this problem is to recertify every single model once in a while, but considering the costs for that it is obviously not a very good solution. So there has to be something better!?

### **3. Fingerprinting and MD5**

Looking for more practicable solutions than recertifying every Petri net model after a certain time, the idea came up to digitally sign the original net – just like a fingerprint. That means every time the net is changed somehow, its fingerprint changes, too.

So now, for the owner of the original fingerprint, it would be as simple as comparing the signatures, which, of course, can be done automatically, and he/she can be sure whether the net is still the original one or not.

After some research and since the problem itself isn't really a new one, MD5 seemed to be the algorithm of first choice.

MD5 (Message-Digest algorithm 5) is a cryptographic hashing function, designed by Ronald Rivest in 1991, which computes a (unique) 128bit digest of a text message of arbitrary length. As soon as one single bit in that message is altered, the entire hash value changes. [1]

Another advantage, that makes MD5 valuable for signing documents, is its speed in computing these hash values, even for very large messages. "The MD5 algorithm is designed to be quite fast on 32-bit machines. In addition, the MD5 algorithm does not require any large substitution tables." [2] A performance analysis can be found in RFC 1810. [3]

The following is a short description of the MD5 algorithm:

First the input message is padded so that its bit-length is a multiple of 512. This is done by adding a "1" to the end of the message and afterwards extending it with "0"s just until the message is 64bits short of this multiple of 512. The remaining bits are filled with a 64bit integer representation of the original message length. (The message is padded in any case even if the original message is already 64bits short of a 512bit-multiple which could imply that the length can be added right away.)

The main algorithm operates on a 128bit state (the later hash value) which is split up into four 32bit words, initialized to certain fixed constants. The padded message is processed in 512bit blocks, each block modifying the state by performing four so called rounds of 16 similar operations each. After all blocks of the message have been processed the 128bit state holds the hash result, usually represented as a 32-digit hexadecimal number. For further details on the constants, used functions or the algorithm itself see definition RFC 1321. [2]

MD5 is widely used, for example to encrypt and store passwords or to ensure that a downloaded file has not been altered.

Throughout the last years there have been several, more or less successful attacks on MD5 which might lead to a security problem in the future. These attacks base on the fact that the hash value, which was formerly assumed to be unique, is not quite that. However, until now it has only been accomplished to do a collision attack that is creating two messages which produce the same (not predefined) hash. Besides, that attack only worked for 1024bit long messages. [4] So for longer messages like in our case, if even computationally feasible at all, it would still take a tremendous amount of time, computing power and analytical human intelligence to find two files with the same hash value and the chances of an accidental collision are said to be close to zero.

Actually more useful would be the ability to create a message producing a certain hash, which gives a potential attacker the possibility to quite easily exchange

documents that have already been signed. Such kind of attempt would be a pre-image attack, but has not been found yet.

In conclusion, using MD5 for signing Petri nets can be seen as a justifiable compromise consisting of low run time and sufficient security. If the security problem will become more severe in future alternative algorithms like SHA-1 (or higher), RIPEMD-160 or others, not yet developed, might be a better choice.

#### **4. Implementation and Integration into SNOOPY**

SNOOPY saves its files in XML format. For reading and writing of these files Xerces-C++ version 2.5.0 of the Apache Software Foundation [5] is used because this library is freely usable and provides all the capabilities needed to easily perform operations on XML data.

For the integration into SNOOPY and also the implementation of SIVE some functions of the wxWidgets library version 2.5.3 (formerly wxWindows) [6] were used. This library had already been used for developing SNOOPY itself, so there was no need to introduce any other new library to this whole software project.

##### **4.1 Signing a SNOOPY-file.**

First the entire XML file, that holds the net, is read into a data model, the so called DOM (Data Object Model), which is done by the Xerces parser. This data model consists of a section containing the nodes of the net and a section containing the edges where each node, edge respectively, may have some layout information, such as position, appearance etc. Basically this model is just a memory representation of the XML format of the file. All XML text nodes, found by the parser, are removed right afterwards because if the file was a proper SNOOPY-file these nodes should only contain ignorable white spaces. This step is needed in order to obtain equality of two identical nets which might only differ in white spaces within their files.

Then the signing itself is done using MD5. There are actually two signatures created, one only for the structure information of the net, like nodes, edges, markings etc., and one for the whole net. This way the user can determine whether only the layout of the net was changed or if there were modifications of essential structural parts. To compute the signature of structure a graphic filter is applied on the DOM that filters all layout information allowing only structure information to be considered.

Finally the two signatures are written to an extra file named *<SNOOPY-filename>.sgn* which is stored in the same directory like the SNOOPY-file. Along with the signatures the original file's name and a time stamp are saved, too.

The integration of this signing function into SNOOPY was done by adding an extra menu item (*Save As and Sign...*) to the file menu. When using this option the current net is saved under an arbitrary file name and afterwards it is automatically signed.

NOTE: If an older signature file already exists it is overwritten!

## **4.2 Verifying a SNOOPY-file**

The verification of a SNOOPY-file in general takes place right after opening it. This is done automatically if there is a corresponding signature file. If not, verification is skipped.

The process itself follows almost the same pattern like the signing. First the DOM of the opened file is created and then the two signatures are computed. The next and also last step is to compare the just computed signatures with the ones saved in the signature file. In case the file signatures are equal the entire net is in its original state. If only the structure signatures match the layout of the net has been modified and if neither pair of signatures matches there are also structural changes.

For implementation details or code samples see chapter 6 and the source code.

## **5. The SIVE-Tool**

SIVE stands for Sign and VErify. This tool has the same functionality as the integrated version. The only difference is that it works as a command line tool apart from SNOOPY, though it is fully compatible. That means files signed with SIVE can be verified by SNOOPY and vice versa.

When using SIVE you have the opportunity to either sign entire directories including subdirectories, certain files or files determined by a wild character mask. Multiple parameters work as well. If no directory is given the tool signs the specified file(s) in the current directory, if no file(s) are given the tool searches the specified directory for all files that have a SNOOPY-file extension (*.spped*, *.sprgraph*, *.spbipart*, *.spgraph*) and signs them.

For verification SIVE is used the same way you just need to add the */v* option before any directory or file specifications.

## 6. Implementation Details and Code Samples

Trying to make as few changes as possible to the original source code of SNOOPY the whole project was developed stand alone. This also gives, though probably never really needed, the opportunity to use it within another project with little or no need of code changes.

The main class of this project is **SP\_Signer**. This class holds the actual object to sign as a member variable that is a DOM representation of the SNOOPY-file. This DOM can either be passed directly through the constructor or be specified by the file's name. In the latter case the DOM is created automatically.

The interface of the class provides the public member functions listed below including their brief descriptions:

**string getSPFileName()**

- get the name of the current object file

**void setSPFileName(string FileName)**

- set a new object file and create its DOM

**string getSignatureOfFile()**

- get the signature of the graph

**string getSignatureOfStructure()**

- get the signature of the graph's structure

**string getSignatureOfFileInSgnFile()**

- get the signature of the graph stored in the signature file

**string getSignatureOfStructureInSgnFile()**

- get the signature of the graph's structure stored in the signature file

**string getTimeStampInSgnFile()**

- get the time stamp stored in the signature file

**void updateSignatureFile()**

- create or update the signature file

**bool verifyFile()**

- verify the graph

**bool verifyStructure()**

- verify the graph's structure

The SP\_Signer class also includes and uses the subclasses **GraphicsFilter** (for filtering layout information of the graph), **ParseErrorHandler** (for simple error handling during the parse process) and **MD5Hasher** (for computing the MD5 hash).

The integration of this project into SNOOPY was done by few modifications in the below listed files as follows:

## **SP\_GM\_DOCMANAGER.H**

*Line 33: → add prototype of event function to header file*

```
void OnSaveSign(wxCommandEvent& p_cEvent);
```

## **SP\_GM\_DOCMANAGER.CPP**

*Line 18: → add necessary include*

```
#include "signing/SP_Signer.h"
```

*Line 25: → define action for menu event (SaveAs And Sign...)*

```
EVT_MENU(SP_MENU_ITEM_SAVE_SIGN, SP_GM_Docmanager::OnSaveSign)
```

*Line 31: → (de-)activate menu entry for certain situations*

```
EVT_UPDATE_UI(SP_MENU_ITEM_SAVE_SIGN, SP_GM_Docmanager::OnUpdateFileSaveAs)
```

*From line 139: → implementation of event function*

```
void SP_GM_Docmanager::OnSaveSign(wxCommandEvent& p_cEvent) {  
  
    SP_MDI_Doc* l_pcDoc = static_cast<SP_MDI_Doc*>(GetCurrentDocument());  
    if (!l_pcDoc) return;  
  
    while (l_pcDoc->GetType() == SP_DOC_COARSE) l_pcDoc = l_pcDoc->GetParentDoc();  
  
    if(l_pcDoc->SaveAs()) {  
        SP_Signer * sgn;  
        try {  
            sgn = new SP_Signer(l_pcDoc->GetFilename().c_str());  
            sgn->writeSignatureToFile();  
            wxMessageBox("The graph has been successfully signed.\n"  
                "The signature file has been written to the current directory.", "Message...");  
        } catch(const char * ex) {  
            wxMessageBox(ex, "Error...", wxICON_EXCLAMATION);  
        }  
        delete(sgn);  
    }  
}
```

## SP\_DEFINES.H

Line 79: → define menu identifier

```
SP_MENU_ITEM_SAVE_SIGN,
```

## SP\_GUI\_MAINFRAME.CPP

Line 113: → create menu entry

```
l_pcFilemenu->Append(SP_MENU_ITEM_SAVE_SIGN, wxT("Save As and Sign..."), wxT("Save the  
current document and sign it"));
```

## SP\_MDI\_DOC.CPP

From line 22: → add necessary includes

```
#include "signing/SP_Signer.h"  
#include "wx/file.h"
```

From line 310: → add verification when opening files

```
bool SP_MDI_Doc::OnOpenDocument(const wxString& p_sFile) {  
...  
    if (wxFile::Exists(p_sFile+".sgn")) {  
        SP_Signer * sgn;  
        try {  
            sgn = new SP_Signer(p_sFile.c_str());  
            if (sgn->verifyFile()) {  
                wxMessageBox(wxString::Format("The graph has not been changed.\n\n"  
                    "Date of signature: %s", sgn->getTimeStampInSgnFile().c_str()),  
                    "Verification result...",  
                    wxICON_INFORMATION);  
            } else if (sgn->verifyStructure()) {  
                wxMessageBox(wxString::Format("The graph's layout has been changed,\n\n"  
                    "but the structure is still unchanged.\n\n"  
                    "Date of signature: %s", sgn->getTimeStampInSgnFile().c_str()),  
                    "Verification result...",  
                    wxICON_EXCLAMATION);  
            } else {  
                wxMessageBox(wxString::Format("The graph has been changed.\n\n"  
                    "Date of signature: %s", sgn->getTimeStampInSgnFile().c_str()),  
                    "Verification result...",  
                    wxICON_ERROR);  
            }  
        } catch(const char * ex) {  
            wxMessageBox(ex, "Error...", wxICON_EXCLAMATION);  
        }  
        delete(sgn);  
    }  
}
```

## **7. Tests and Performance**

### **7.1 Testing**

The testing of this project was done in two stages.

The first was testing the individual components by itself, in particular the MD5 hashing and the signing.

The implementation of MD5 was tested with the standard test messages of the official definition [RFC 1321] and furthermore successfully crosschecked for several files using the included reference implementation. The only limitation of this project's MD5 implementation is that it works only for text files properly, which, of course, includes the XML file format. That is because the class for hashing uses a string to hold the message which means if using other file formats the original message might contain NULL-characters somewhere, which would terminate the string preemptively and therefore the result would be a different hash. This fact was discovered while testing but it has actually no limiting effect on the project, so it hasn't been fixed yet. In case someone wants to use the MD5 class for another project he/she should be aware of that.

The tests of the signing class included the check for proper layout filtering and DOM normalization by comparing the results with the original files and the check for proper error handling by triggering all the possible exceptions. The file I/O, especially correct reading and writing of the signature files, was also tested.

The second stage was the integration test. It was done by playing through the most common use cases for this feature, applying several changes to the signed graphs and afterwards checking correct recognition by the program. The mentioned changes were done through the program itself and also by modifying the file manually.

In conclusion the program has passed all applied tests successfully, in other terms it worked as expected.

### **7.2 Performance checking**

Performance was only checked for the implementation of the MD5 algorithm, firstly because it differs from the reference implementation, for which there are already performance reports [3], and secondly because it was expected to be the most time consuming routine. For that a dummy SNOOPY-file was created. Originally it contained 600 places, 300 transitions and 1200 edges, which was thought to be already quite big for a Petri net. That file had a size of about 2.5MByte. Doing the hash on it was done in fractals of a second, as expected. Then the file was successively extended up to about 50MByte. That size conforms to 15,000 places, 7,500 transitions and 30,000 edges in the test file. For this net the hash was done with a noticeable delay but still in less than one second. Overall it can be stated that there won't be any performance problems while signing or verifying SNOOPY-files.

## 8. References

- [1] **General information on MD5**  
<http://en.wikipedia.org/wiki/Md5>
- [2] **[RFC 1321] Definition of the MD5 Algorithm; Apr '92**  
R. Rivest; MIT Laboratory for Computer Science and RSA Data Security, Inc.  
<http://rfc1321.x42.com>
- [3] **[RFC 1810] Report on MD5 Performance; Jun '95**  
Joe Touch, Information Sciences Institute  
<http://rfc1810.x42.com>
- [4] **Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD; Aug '04**  
Xiaoyun Wang, Dengguo Feng, Xuejia Lai and Hongbo Yu  
<http://eprint.iacr.org/2004/199.pdf>
- [5] **Xerces-C++, a validating XML Parser in a portable subset of C++**  
The Apache XML Project  
<http://xml.apache.org>
- [6] **wxWidgets, a portable C++ and Python GUI toolkit**  
Julian Smart, Robert Roebing, Vadim Zeitlin, Robin Dunn, et al  
<http://www.wxwindows.org>
- [7] **Microsoft Developer Network**  
Microsoft Corporation  
<http://www.microsoft.com/msdn>

**NOTE:** Platform for developing testing and performance checking was a Notebook with AMD Athlon 64 (~798MHz), 512 MB RAM and operating system Windows XP. Further portability, i.e. under Linux, is intended but has not been tested.