Concurrent Architecture for Automated Malware Classification

Timothy Daly
Software Engineering Institute
Carnegie Mellon University
daly@cert.org

Luanne Burns
Johns Hopkins University
Applied Physics Lab
luanne.burns@jhuapl.edu

Abstract

This paper introduces a new architecture for automating the generalization of program structure and the recognition of common patterns in the area of malware analysis. By using massively parallel processing on large malware program sets we can recognize common code sequences, such as loop constructs, if-then-else structures, and subroutine calls. We can also recognize common subroutine sequences. The Concordia architecture generalizes the recognized elements so they can be collected into invariant forms. The invariant forms can be used by the analyst to understand the program being analyzed. The invariant forms can also be used to classify large numbers of programs automatically.

Motivation

Current practice in malware analysis uses our most expensive resource, namely the analyst, to analyze programs one at a time. The large volume of malicious programs is rapidly overwhelming our ability to analyze malware in a timely manner. The analyst has few tools to automatically classify a particular program into useful categories, so as to understand important properties of malware evolution and trending. Thus each new program analysis often "starts from scratch." When starting from scratch on a new program the analyst has little ability to recognize general structure such as standard library code, sorting and searching routines, and common machinery like command line option handling.

We need to restructure our approach to handle the volume of incoming programs, well over 5000 per day. Given this large catalog of malicious programs, we want to use it as a basis for analysis. In particular, we would like to be able to achieve two goals. The first goal is "bottom up" in that we want to recognize, extract, and generalize common elements in a new program so the analyst can focus on the new material. The second goal is "top down", we would like to capture and use the knowledge of past programs so analysts can benefit from each other's work. In essence, Concordia represents a new type of reverse engineering [3] targeted to analysis and classification of malicious code.

Background

This work draws on three streams of research, supervised learning, Hawkin's cortex architecture, and Google's Map/Reduce architecture.

Concordia uses unsupervised learning at the lower layers; specifically we use a form of non-statistical hierarchical clustering [6]. The input stream can contain tags that associate program fragments with their original source. Small program fragments are clustered at higher layers to group common elements. The metrics used for clustering are based on Function Extraction (FX) technology for automated software behavior computation [2,7,9,10,] as is explained in the body of the paper.

At the upper layers learning is supervised. We use information gained from the analyst to label clusters of code sequences. If a cluster of program fragments is known to have the same semantics, for instance, being an iterated-conditional-swap and the analyst labels one of the elements in the cluster as a *sort* then all code sequences in that cluster are sorts. This captures the experience from one analyst so it can be used by all.

Concordia uses the structural organization found in Hawkin's [5] explanation of the cortex of the brain. Hawkins conjectures the function of each cortex layer in a feed-forward, feed-back mechanism to construct invariant representations from sensory data. This work is used as a guide to partition Concordia into manageable semantic layers. The definition of the function of layers in Concordia is presented in the body of the paper. The invariant we strive to achieve is to recognize a class of malicious program (e.g.

978-0-7695-3869-3/10 $26.00

VMProtect) no matter what particular instruction sequences are used.

Finally, since the malicious code task needs to handle extremely large volumes of data we arrange the system to work in parallel at every level. Map/Reduce [1] and Hadoop [17] are the models for the control organization.

Novelty

Concordia introduces the notion of "chipping" programs which reduces programs to primitive sequences. Using FX technology, we can extract the exact semantics of these chips despite different instruction order. FX allows us to cluster based on program semantics. Bottom up semantic clustering combined with top down semantic tagging gives us the ability to recognize program behavior such as sorting.

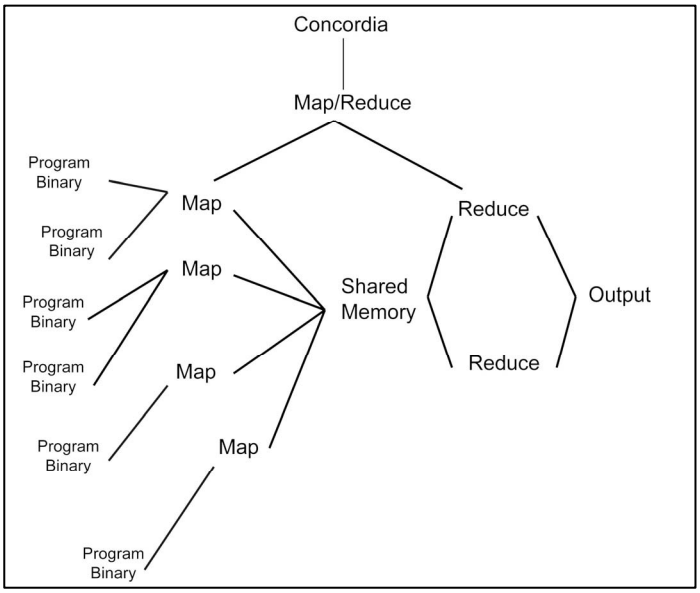

Figure 1: Google Map/Reduce

Architecture

The Concordia design is fully parallel so its implementation can handle large volumes of data. The top level control structure follows the Google Map/Reduce architecture [1] as shown in Figure 1.

Concordia expects a single program as input. Each stage of the processing of that input can be done in a parallel fashion on program chunks. Since there are no dependencies between the programs being processed, many thousands of programs can all be analyzed in parallel.

Map Process

We refer to the initial processing of a program as "wood chipping." Basically, each program entered into the repository gets mapped into chips that are two instructions long, where each chip overlaps the prior one. Another process breaks the same program into overlapping sequences of three instructions. Yet another process breaks the same program into overlapping sequences of four instructions. So you'll see instructions (1,2,3,4), then (2,3,4,5), then (3,4,5,6), and so on, as the chips of length four. Thus, a single input program gets processed multiple times in parallel to create these instruction sequences as shown in Figure 2.

This "chipping process" we have described is tailored to virus program analysis. If the goal was to process other massive data sets, for example, the images of bubble chambers for particle searching, the chipping process might slice individual particle tracks out of the image data.

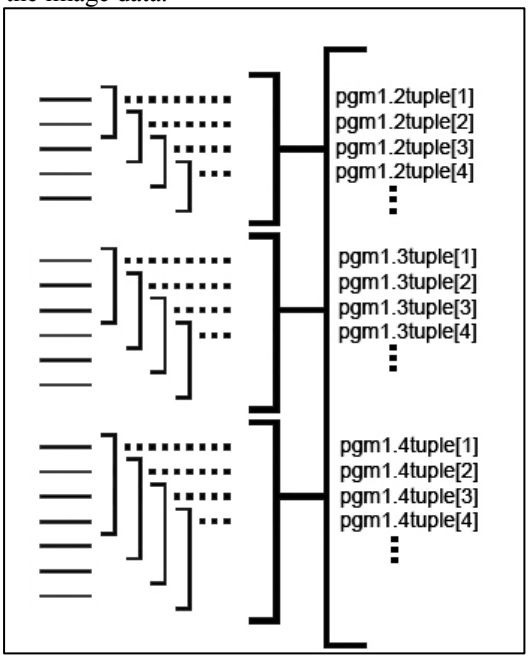

Figure 2: The Repository Cords

Cords

All of the information about a chip is kept in an associated data structure called a cord. A cord also contains such information as tags. These tags can associate individual chips with their source. Cords contain offset information from the original source, membership pointer to equivalence classes, and

results of equivalence metrics such as the conditional concurrent assignments from Function Extraction. We might wish to know that a piece of code came from a library like glibc so we can recognize it later. Chips that fall into the equivalence class with library code would have the same Function Extraction semantics and thus perform the same function despite obfuscation.

Reduce Process

The Reduce processing is broken up into 6 stages. The early stages process individual program chips in an independent fashion. Later stages look for common elements and are less parallel. However, the earlier stages combine and filter their data so the later stages have less data to process.

Reduce Process Stages

The Concordia reduce process has 6 stages:

1. Recognition
2. Equivalence
3. Classification
4. Generalization
5. Invariance
6. Prediction

These stages gradually move from raw input to more generalized input. We'll discuss the stages one at a time.

Stage 1: Recognition

The first step is to make the program chips unique. Thus when we find a particular code sequence in 5000 programs we collect them together. Clearly a program text that occurs frequently is of great interest.

Duplicate program sequences can occur frequently because library code is loaded at fixed offsets, and common program startup code such as the C runtime stub, **CRT.o** will occur in every C program.

The cords are all bound together so further stages see all of these duplicate chips as a single item but the individual cords can still carry information about the original source of the chip.

Stage 2: Equivalence

The next step is to group the chips into equivalence classes. In general, it is more likely to see pieces of programs which are equivalent but not the exact same code, due to variations in addresses, statement order, or optimizations. The notion of "equivalence" is vital to the process since this is where we discover that apparently distinct program fragments really have the same meaning. We will describe two ideas briefly to illustrate the kind of equivalence metrics of interest:

- Lambda reduction
- Function Extraction

With lambda reduction, a plane is formed. The X-axis of the plane will represent state locations in the machine. The Y-axis represents state changes over time. We are looking for equivalent patterns in space and/or time.

Equivalence: Lambda reduction

Consider the X-axis. We assign main memory locations starting at 0, up to the limit of memory. We assign other storage such as disk space with higher numbers on the X-axis. So memory location 100 would be at X=100.

We use the negative limb of the X-axis to locate registers and other machine state. So the EAX register might be at -12, EBX at -16, etc.

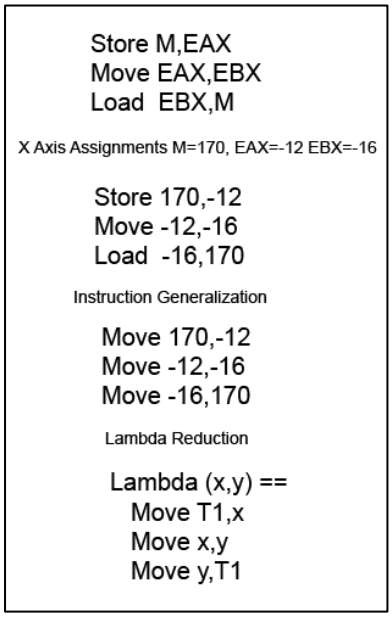

Figure 3: Lambda reduction of swap

Using this linear layout of memory we can look at a simple piece of code to swap to values.

Here we perform three steps. Step one is the renaming of storage locations into X-axis locations. The special case of registers is removed. Step two renames the machine instructions into move instructions which removes the machine specific instructions. Step three involves a lifetime analysis to remove the temporary variable and generate a lambda form. This "lambda reduction" technique is illustrated in Figure 3.

Equivalence: Function Extraction

The second equivalence idea is to use Function Extraction technology [2,7,9,10]. Here the idea is that we compute the actual behavior of the program chip and express the behavior using "conditional concurrent assignments" (CCAs). The Function Extraction behavior is independent of the program specifics and expresses what the CPU will actually compute. This can be used to recognize, for example, that the four code sequences shown in Figure 4 all perform a "swap" operation:

```
a=x;
x=y;
y=a

x=x-y;
y=x+y;
x=y-x

x=x^y;
y=x^y;
x=x^y

a=x-y;
if (a>0)
  then {x=x-a; y=y+a}
  else {x=x+a; y=y-a}
```

Figure 4: Function Extraction of swap

The end result of the equivalence process is to gather together code sequences that are equivalent under some user-defined metrics. At this point we have taken the various program chips and made them into uniform equivalent groups.

Stage 3: Classification

Now we consider the structural semantics of the cords and their associated chips. This phase constructs classifiers for program sequences. Classifiers are active elements that "bind" to their specific site. We would like to consider the repository to be similar to a pool of enzymes. The analyst "dips" a new program into the pool and when it is extracted, there will be parts of the program with cords attached. These cords "bind" to the equivalent structures of code found in the analyst's program. To do this we construct patterns that represent the general case of the equivalent program.

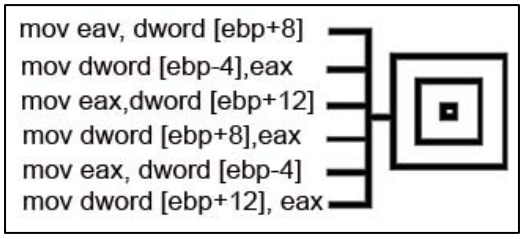

Figure 5: Swap cord classifier

The swap classifier shown in Figure 5 has a binding site for each of the important pieces of code that need to be part of the swap sequence.

The conditional classifier shown in Figure 6 has a binding site for each important piece of code. Notice that there are sections of the conditional which could contain arbitrary code sequences.

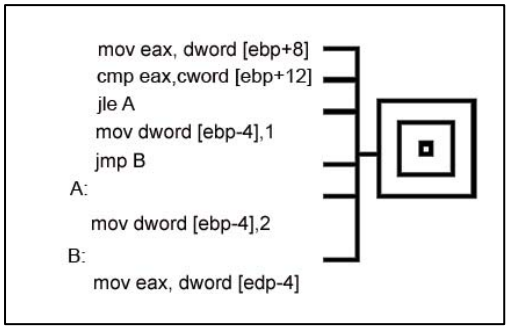

Figure 6: Conditional cord classifier

The iterator classifier shown in Figure 7 has a binding site for each important piece of loop control code.

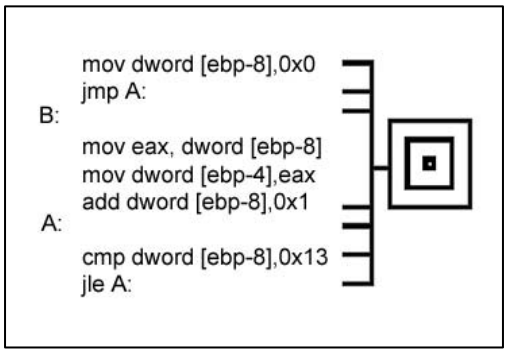

Figure 7: Iterator cord classifier

These active classifiers work similar to enzymes, looking for, and attaching to compatible binding sites represented by compatible code sequences. The active code classifiers ignore code which does not involve their particular pattern. Conditionals, for instance, would involve the body of the "then" and "else" cases in a standard if-then-else program statement. The classifier is matching control structure and would ignore those statements, leaving them to other cords to match.

In general, these matches can nest, as they do in the case shown above where the swap sequence is nested within the conditional sequence and that is nested within the iterator. Cord Enzymes are illustrated in Figure 8.

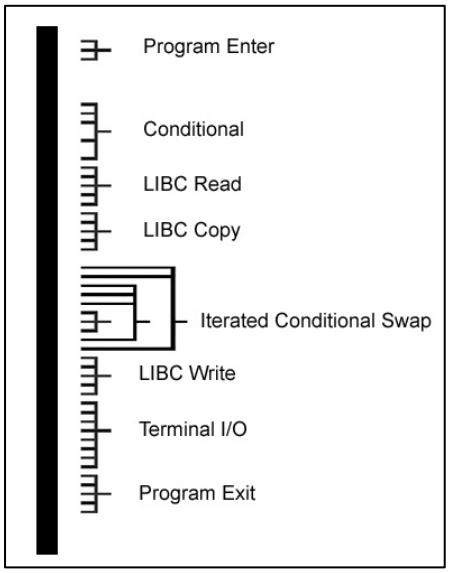

Figure 8: Cord Enzymes

Stage 4: Generalization

In the generalization phase we move from structural recognition to the semantics of the program chunks. When the repository is initially populated we do not know anything specific in this phase. We can infer from repetition that certain kinds of forms occur frequently. We can infer that there is something interesting about them. But we need information either from the analyst or from semi-automated tools to help name the structures we recognize.

A semi-automated method of recognizing structures would be to feed the repository with "tagged" pieces of program code. For instance, the standard C library routine **strcpy** would be tagged at chipping time. That code fragment would have an associated cord and pattern that could be recognized and tagged in every other program that uses **libc.**

An analyst or repository maintainer can recognize and tag certain common patterns manually. For instance, notice that in Figure 8 above we have the nested set of cords:

Iterated
Conditional
Swap

This is almost certainly a sort routine. Other common patterns might be

Search == Iterated Conditional Compare
Initialize == Iterated Unconditional Assignment
Map == Iterated Function Application

A whole series of other recognizable patterns might include filter, map, reduce, collect, fold, take, drop, broadcast, collect, replicate, mapcar, maplist, fork, and join.

Virtual Instructions

In the virus area, there is a technique of virus obfuscation that involves using a virtual machine instruction set. The idea is to make up a sequence of instructions for a virtual processor that perform some operations, such as **add, load, shift**, etc. and then compile the actual instructions into obscure code sequences using these virtual instructions.

Concordia can recognize that such code sequences exist, find their boundaries, and use Function Extraction to compute their behavior. We can

recognize the common **add** subsequence. However, we cannot generate the label "add." That would have to come from an analyst. Once the label is recognized in a single program it will be applied to all programs that also contain the **add** subsequence. This naturally leads us to the next topic of invariance.

Stage 5: Invariance

At this point we can recognize chunks of a program and we can leverage information gathered by an analyst to inform us of program features.

The invariance layer is directed toward whole program information and features rather than parts of the program.

Consider the case of a particular virus that has a copy of the C library routine **strcpy.** What can we infer from this code? It turns out that various compiler switches can wildly affect the size of the code that will be generated. The various Concordia phases add pieces of information that turn out to be very useful for the analyst.

First, the map process (chipping) phase can be used to chip the standard C library code with various compilers and compiler settings. This tag information would be added before chipping. Each of these cords associated with the chip would carry the tag that provides information about the compiler options used.

The Function Extraction technology in the equivalence phase can reduce these wildly varying code sequences to a single equivalence class based on their behavior. Thus, all of the **strcpy** code blocks could be recognized even if we didn't know that it was named **strcpy.**

The classification cord enzymes would recognize the **strcpy** code sequence and since we have an associated tag it can be properly labeled.

The generalization phase normally generates random names for sequences. However, if we know certain patterns and their associated names (e.g. swap) we can better recognize that these combine into larger patterns such as "sort."

The invariance phase now (potentially) knows what compiler and what command line options were used to compile this program. This "whole program" information would be presented to the analyst.

Another example of invariance inference involves the use of the virtual instructions mentioned above. If we can find the code sequences for virtual instructions we can conclude that the obfuscation mechanism was used for this program. In particular, if the analyst has tagged the sequences such as add, load, shift, etc. we can know what the program is doing. Better yet, if we can find the tool online that was used to create this virtual obfuscation we could identify what was used to obscure the virus.

A third example comes from the data structures within the program. The C++ language uses C-structs that contain function pointers and data that represent the object. Objects that inherit from that object have a backward pointer to the original struct. By observing the data layout we can infer that this program follows the hierarchical inheritance paradigm.

Calling sequences that incorporate these struct pointers as the first item in a calling sequence are very likely to be object-oriented programs.

Other indicators within the program imply the use of particular design patterns. If all of the subprograms are accessing a common queue we can recognize a boss and workers pattern.

Ultimately we can compare and contrast programs to find behavior which is unexpected. For example, if we tag and chip a copy of Windows Notepad into the repository and we find a version of the same program, we might note that the "virus" form of the program has code to access the network which does not exist in the original code. In this way we can highlight behavior identified by function extraction to be unexpected and worthy of attention.

The ultimate goal of the invariant phase is to provide high level general information to the analyst. Ideally we would be able to say that "this code is Windows Notepad *with additional network behavior.* That is, we have highlighted to the analyst that we have found uncommon behavior based on prior experience with Notepad.

Stage 6: Prediction

The repository represents the accumulated experience of analyzing many viruses from the catalog along with the accumulated wisdom of the analysts.

The prediction phase interacts with the analyst to query and extract "similar" programs, or programs with features in common with the program under

study. Thus, the analyst could do a SQL-style SELECT from the repository for all programs which were compiled with GCC 4.2, using the virtual instruction set, and accessing the network. On finding these other programs in the repository, it might be possible to conclude this is a duplicate.

One key advantage of Concordia is that it can adapt to new viruses in a dynamic way. A new virus will form its own cluster. This will provide an early warning system for new threats, which is a vital tool for concentrating effort on emerging problems.

In addition, Concordia gathers knowledge gained, so any annotations done by the analyst can be pushed into the repository and the information tag will follow the cord all the way back through the process. In this way the system gradually learns what is known and shares this knowledge with all users.

Implementation

The whole design of Concordia is massively parallel, both at the level of single program analysis and at the level of multiple program analysis.. In order to fully leverage this parallel nature, an implementation needs to be very careful about the details to ensure that there are a minimum number of points where shared state occurs.

Portions of Concordia have been implemented in a system to detect a particular virus that uses polymorphic packing in order to obfuscate the program. The implementation found 132,000 instances of this technique in 3.9 million programs in a malware catalog [17]. The cluster metric is based on the first 100 bytes of program code. This gives us a practical demonstration of clustering based on code chunks. The polymorphic packing scanner program was not done in parallel. One scan of the catalog took about 10 minutes. Given the number of newly arriving programs this would not be able to recognize more than one type of virus in near-real time. As more recognizers are added the work would have to be done in parallel if we are to perform recognition and classification in near real time.

Benefits

Concordia is designed to handle the flood of malware programs. It automatically extracts program structure and creates cords to remember structure it has seen. Concordia automatically classifies at every level of the cord so low level and high level queries can be performed. It applies what it has learned in general to a particular program under study by an analyst and automatically adapts to new kinds of programs that occur in the input stream over time which improves our ability to recognize and handle new threats.

Summary

We have introduced a new architecture for automating the generalization of malware program structure and the recognition of common patterns. By using massively parallel processing on large program sets, we can recognize common code sequences, such as loop constructs, if-then-else structures, and subroutine calls. We can also recognize common library subroutine sequences.

The Concordia architecture generalizes the recognized elements so they can be collected into invariant forms. These invariant forms can be used by the analyst to understand the program being analyzed. The invariant forms also can be used to automatically classify large numbers of programs and dynamically recognize new threats in very short time scales.

References

[1] Dean, Jeffrey, Ghemawat, Sanjay, "MapReduce: Simplified Data Processing on Large Clusters", *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 12/04

[2]Collins, R., Walton, G., Hevner, A., Linger, R., "The CERT Function Extraction Experiment: Quantifying FX Impact on Software Comprehension and Verification", Technical Note CMU/SEI-2005-TN-047, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005.

[3]Eldad, Eilam, *Reversing: secrets of reverse engineering,* John Wiley and Sons, 2005, ISBN 0764574817, 9780764574818

[4]Gallagher, J.R. Lyle, "Using Program Slicing in Software Maintenance", *IEEE Transactions on Software Engineering*, *Vol. 17, No. 8, pp. 751-761, Aug. 1991*

[5]Hawkins, Jeff, *On Intelligence*, Henry Holt and Company NY, 2004, pp106-177

[6]Hierarchical Clustering *SAS/STAT (R) 9.2 User's Guide* http://www.sas.com

[7]Hevner, A., Linger, R., Collins, R., Pleszkoch, M., Prowell, S., Walton, G., "The Impact of Function Extraction Technology on Next-Generation Software Engineering", Technical Report CMU/SEI-2005-TR-015,

Software Engineering Institute, Carnegie Mellon University, July 2005.

[8]Linger, R., Mills H., Witt, B., *Structured Programming: Theory and Practice*, Addison-Wesley, Inc., 1979.

[9]Linger, R., Pleszkoch, M., "Improving Network System Security with Function Extraction Technology for Automated Calculation of Program Behavior," *Proceedings of the 37th Annual Hawaii International Conference on System Science (HICSS35),* Hawaii, IEEE Computer Society Press, Los Alamitos, CA, January 2004.

[10]Linger, R., Pleszkoch, M., Burns, L., Hevner, A., Walton, G., "Next-Generation Software Engineering: Function Extraction for Computation of Software Behavior," *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS40)*, Hawaii, IEEE Computer Society Press, Los Alamitos, CA, January 2007.

[11]Lyle, J.R., Gallagher, K.B., "A program decomposition scheme with applications to software modification and testing," *Proceedings of the 22th Annual Hawaii International Conference on System Sciences (HICSS22)*, Hawaii, IEEE Computer Society Press, Los Alamitos, CA, January 1989.

[12]Mili, A., Daly, T., Pleszkoch, M., Prowell, S., "Next-Generation Software Engineering: A Semantic Recog nizer Infrastructure for Computing Loop Behavior," *Proceedings of Hawaii International Conference on System Sciences (HICSS41)*, Hawaii, IEEE Computer Society Press, Los Alamitos, CA, 2007.

[13]Pleszkoch, M., Hausler, P., Hevner, A., Linger, R. "Function-Theoretic Principles of Program Understanding," *Proceedings of the 23rd Annual Hawaii International Conference on System Science (HICSS23),* Hawaii, IEEE Computer Society Press, Los Alamitos, CA, January 1990.

[14]Pleszkoch, Mark, Cohen, Cory, "Detection of the Allaple Polymorphic Packer" CERT Software Engineering Institute Internal report 2009

[15]Poore, J., Mills, H.,Mutchler, D. "Planning and Certifying Software System Reliability", *IEEE Software, Vol. 10, 1993.*

[16]Prowell, S., Trammell, C., Linger, R., Poore, J. *Cleanroom Software Engineering: Technology and Practice,* Addison Wesley, Reading, MA, 1999.

[17]White, Tom, *Hadoop: The Definitive Guide*, O'Reilly Media Inc, CA 2009