Concordia: A Google for Malware

Timothy Daly
Software Engineering Institute
Pittsburgh, PA
daly@cert.org

Luanne Burns
Software Engineering Institute
Pittsburgh, PA
luanne.burns@jhuapl.edu

ABSTRACT

This paper introduces a new architecture for automating the generalization of 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 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.

Categories and Subject Descriptors

H.4 [**Program Understanding**]: Function Extraction; D.2.8 [**Software Engineering**]: Program Understanding—*Concordia, Function Extraction, Machine Learning, Malware*

Keywords

Concordia, Function Extraction, Machine Learning, Malware

[1]

1. THE PROBLEM

The large volume of malicious programs is rapidly overwhelming our ability to be effective. The analyst has few tools to automatically classify a particular program into reasonable categories. Thus each new program analysis "starts from scratch". When starting from scratch on a new program the analyst needs tools to recognize general structure such as standard library code, sorting and searching routines, and common machinery like command line option handling. While these might be obvious in ordinary binaries, the malware writers go to great lengths to obfuscate this information.

2. THE NEED

We need to restructure our approach to handle the volume of incoming programs. We need to operate in parallel to recognize, extract, and generalize common elements. The common elements can be used to automatically classify programs by many different metrics such as prioritizing the programs by attack type. The common elements can be used by the analyst to automatically recognize most of the code forms in a particular program, reducing the time and effort needed to understand a single program.

3. ARCHITECTURE

The Concordia design is fully parallel so the program can handle large volumes of data. The top level control structure follows the Google MapReduce architecture. [13, 18, 19]

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.

We refer to the initial processing of a program as "wood chipping". Basically, each program entered into the repository gets broken up 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 there will be 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 program gets processed multiple times in parallel to create these instruction sequences.

The same chipping process can be applied in parallel to multiple programs, creating vast quantities of data in the repository.

4. THE SIX LAYERS OF PROCESSING

Processing is broken up into 6 layers [6, 7]. The lower layers process individual program chips in an independent fashion. Later layers look for common elements and are less parallel. The lower layers combine and filter their data so later stages have less to process.

4.1 Recognition

The first step is to make the program chips unique. When we find a particular code sequence in multiple programs we collect them together. Clearly a program text that occurs frequently is of great interest. Duplicate program sequences can happen 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. These code sequences are grouped together but information about the original source of the information is kept.

4.2 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 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 mention three techniques here, lambda reduction, function extraction, and perceptrons.

4.2.1 Lambda Reduction[2]

Lambda reduction is defined in terms of a plane. The horizontal axis in the plane assigns numbers to state locations in the machine. For instance, the registers can be numbered from zero toward negative infinity. The memory can be numbered from zero to positive infinity, followed by the disk locations as higher numbers. The vertical axis is time.

We can take a code sequence and replace the registers and memory locations with numeric values. We can then represent most operations as simple moves from number to number which abstracts away machine specific operations like load and store. Finally we can perform a lifetime analysis to define the scope of location lifetimes. These, and other compiler-style techniques, abstract away machine details.

4.2.2 Function Extraction[10]

Function extraction (FX) technology under development by CERT is directed to automated computation of the full functional behavior of programs. The FX process begins by expressing the semantics of each machine instruction in an input program as a conditional concurrent assignment (CCA). The CCA captures all of the functional behavior of each instruction including side-effects like flag settings.

The program is structured to eliminate control flow obfuscation and creates a hierarchical algebraic structure of sequence, if-then-else, and while-do structures. The CCAs for each instruction are functionally combined to create new combined CCAs that capture the net behavior of the structure.

In this way we build up a "behavior catalog" that shows the as-built behavior of the program based on the ground truth of the machine semantics. This has the effect of removing obfuscation techniques, such as inserting long sequences of instructions that do nothing but confuse the analyst. Behavior computation reveals that these instruction sequences have no net effect and can be eliminated.

Of special interest in the Concordia context is the "many implementations, one function" property of behavior computations. There are many procedures that can be written to implement a function. Function extraction reduces them to a single non-procedural form, thereby revealing common functions effects across different implementations.

4.2.3 Perceptrons [1, 5]

Perceptrons recognize pattern templates in code sequences. These are typically used in vision processing but we generalize the idea here for code patterns. We want to recognize if-then-else patterns, loops, subroutine entry and exit, argument processing, and other common forms.

We also use the perceptron idea to recognize memory shapes for data structures such as arrays, linked-lists, and structs.

Higher level patterns recognize code sequences such as a loop that does unconditional assignment , a loop that does a compare and swap , or data structures such as data combined with function pointers in structs used as the basis for object oriented programming.

4.3 Classification

Machine learning techniques [4, 9, 11, 12, 15, 16, 17] are applied to group the code sequences into sets we call "enzymes". These sets have a central form of behavior that represents the general case. Both unsupervised and supervised learning techniques are applied to group the code sequences.

The unsupervised learning is the accumulation of the code clusters from processing the malware catalog through the lower layers. Various distance metrics are used to separate or group the code piles.

Supervised learning involves both bottom-up and top-down processes. In the bottom-up learning the analyst feeds specially tagged program data into the normal chipping process, for instance, a copy of the glibc subroutines. Code that groups with a known subroutine is claimed to be a variant of the tagged code.

For top-down learning the analyst has a "cord picture language" that enables the construction of a disjunctive normal form pattern. The recognizers are combined with filters and the results are joined into a single pattern. This pattern is added to the system and used for classification.

4.4 Generalization

The generalization layer tries to bridge the gap between recognizing **that** something exists as a pile and recognizing **what** that pile might be. Here we depend on the analyst to provide these names either through tagging or direct manipulation. We would find names of code patterns such as [4, 9]:

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

4.5 Invariance

At this point we can recognize chunks of a program and we can leverage information gathered by one or more analysts to inform us of program features. The invariance layer is directed toward whole program information and features rather than parts of programs.

Consider the case of a particular virus that uses 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. By inserting multiple copies of **strcpy**, each tagged with the compiler switches, we can recognize which collection of switches were used. From that we can infer the original compiler information.

Another example is the recognition of the so-called **VM-Protect** instructions. These are instructions, such as `add`, that are written for a virtual machine and then simulated on the real machine. Since the analyst does not know the virtual machine instructions there is no easy way to recognize what is going on. Concordia will group these virtual instructions together. If any single virtual instruction sequence is recognized we now know a great deal about the program under study.

4.6 Prediction

The Concordia repository represents the accumulated experience of malware 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 an 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.

A general pattern language of recognizers, filters, and joins is provided to allow the analyst to construct disjunctive normal form queries and patterns. This allows the analyst to search, extend, and organize the information in the repository.

In this form Concordia becomes a "Google for Malware". We are applying techniques similar to what Google uses to recognize personal preferences as a basis for their direct marketing campaign[14].

5. BENEFITS

Ultimately, given a new unknown malware program, Concordia provides a body of knowledge and experience to automatically extract the maximum amount of information based on past malware. It is expected that this tool will greatly reduce the time necessary to identify a threat.

The goal is to be able to process new malware programs in real time. Most of the heavy lifting is a one-time process to populate the repository.

Concordia adapts to what it sees. This is particularly useful in the near future as the malware threat is changing from general purpose attacks to company specific attacks. Despite that change there are only so many different techniques that can be used and Concordia will be attuned to most of them. Unlike signature based matchers, this makes it very difficult to create a new program that would pass review.

Due to its massively parallel design and the large catalog of already captured malware, Concordia has the capability to perform the analysis of a new program in near real-time. Programs that appear malicious can be quarantined for further analysis.

We see Concordia used as a real-time triage of incoming programs; an ever-improving front-line sentry.

6. REFERENCES

[1] Bongard, M. "Pattern Recognition", 1970 Spartan Books ISBN `0-87671-118-2`

[2] Daly, Timothy and Burns, Luanne "Concurrent Architecture for Automated Malware Classification" Hawaii International Conference on System Sciences 43 Jan. 5-8, 2010

[3] Dean, J. and Ghemawat, S. "MapReduce: Simplified data processing on large clusters" Proc. Sixth Symp. on Operating System Design and Implementation San Fransisco, CA Dec.6-8 2004 `http://labs.google.com/papers/mapreduce.html`

[4] Dietterich, Thomas G. and Michalski, Ryszard S. "Learning to Predict Sequences" pp 63-106 from "Machine Learning", Vol 2 Morgan Kaufmann Publishers, ISBN `0-934613-00-1`

[5] Duda, Richard and Hart, Peter "Pattern Classification and Scene Analysis" pp10-256 Wiley Interscience 1972 ISBN `0-471-22361-1`

[6] Eggermont, Jos J. "Learning - The Neocortex" pp246-279 from "The Correlative Brain" Springer-Verlag 1990 ISBN `0-387-52326-X`

[7] Hawkins, Jeff "How the Cortex Works" pp106-176 from "On Intelligence" 2004 Henry Holt and Company ISBN `0-8050-7853-3`

[8] "Hierarchical Clustering SAS/STAT (R) 9.2 Users Guide" `http://www.sas.com`

[9] Kodratoff, Yves and Ganascia, Jean-Gabriel "Improving the Generalization Step in Learning" pp215-244 from "Machine Learning", Vol 2 Morgan Kaufmann Publishers, ISBN `0-934613-00-1`

[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] MARS Marsland, Stephen "Machine Learning. An Algorithmic Persepctive" Chapman and Hall/CRC 2009 ISBN 978-1-4200-6718-7

[12] Ng, Andrew. "Machine Learning (CS229)" `http://www.youtube.com`

[13] Perrott, R.H. "Parallel Programming" Addison-Wesley Publishing, 1987 ISBN `0-201-14231-7`

[14] http://code.google.com/p/protobuf

[15] Michalski, Ryszard S. "A Theory and Methodology of Inductive Learning" pp83-134 from "Machine Learning", Vol 1 Morgan Kaufmann Publishers, ISBN 0-934613-09-5

[16] Michalski, Ryszard S. and Stepp, Robert E. "Learning From Observation: Conceptual Clustering" pp331-364 from "Machine Learning", Vol 1 Morgan Kaufmann Publishers, ISBN 0-934613-09-5

[17] Tou, Julius and Gonzalez, Rafael "Pattern Recognition Principles" Addison-Wesley Publishing 1974 ISBN 0-201-07586-5

[18] White, Tom "Hadoop: The Definitive Guide" O'Reilly 2009 ISBN 978-0-596-52197-4

[19] Zhao, Jerry and Pjesivac-Grbovic, Jelena "MapReduce - The Programming Model and Practice" http://research.google.com/archive/papers/mapreduce-sigmetrics09-tutorial.pdf